AFRL-RI-RS-TR-2018-185

ENERGY EFFICIENT SIGNAL PROCESSING ON 3D MEMORY INTEGRATED MULTI-CORE PLATFORMS

UNIVERSITY OF SOUTHERN CALIFORNIA

AUGUST 2018

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* MAY 2018	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* JUL 2015 – FEB 2018

4. TITLE AND SUBTITLE ENERGY EFFICIENT SIGNAL PROCESSING ON 3D MEMORY INTEGRATED MULTI-CORE PLATFORMS	**5a. CONTRACT NUMBER** N/A
	5b. GRANT NUMBER FA8750-15-1-0185
	5c. PROGRAM ELEMENT NUMBER 62788F
6. AUTHOR(S) Viktor K. Prasanna	**5d. PROJECT NUMBER** 95SB
	5e. TASK NUMBER US
	5f. WORK UNIT NUMBER CA
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Southern California, 3740 McClintock Ave. EEB 200, Los Angeles, CA 90089-2562	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITB, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2018-185

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Air Force Research Laboratory (AFRL) is developing a secure processor that integrates a multicore logic layer with vertically stacked DRAM. The chief advantage of such 3D Memory Integrated architectures is the large amounts of memory accessible by the on-chip logic via high bandwidth vertical interconnects. While the energy cost of block memory access can be well defined in such a platform, there is currently no method of optimizing the memory accesses of a complex algorithm and its impact on energy consumption in platforms with 3D stacked memory. In particular, the high bandwidth connection between logic and memory in 3D architectures has created an opportunity to redesign the conventional processor-memory cache hierarchy with potentially significant effect on both kernel performance and overall energy efficiency. Mapping applications to this target platform is therefore complex due to the large number of architectural features and their complex performance-energy-efficiency tradeoffs. In this project, we developed a performance model of the Target 3D Memory Integrated Multicore Platform that can be used for evaluating the energy-efficiency and performance-energy tradeoffs of specific signal processing algorithms as mapped onto this target platform. We then demonstrated the efficiency of this framework by mapping representative signal processing kernels and generating design curves describing the trade-off between energy efficiency and algorithm performance. Our results enable the principled and practical exploration of parallel algorithm performance and energy-efficiency and understanding of the impact of architectural design choices on performance-energy tradeoffs, which is central to the adoption of 3D memory, centered platforms in C4ISR applications.

15. SUBJECT TERMS
Signal processing on 3D memory, 3D memory integrated architecture, 3D performance Model, 3D Memory Performance-Energy Trade-off

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** THOMAS E. RENZ
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	42	**19b. TELEPHONE NUMBER (*Include area code*)**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. SUMMARY

The Air Force Research Laboratory (AFRL) is developing a secure processor that integrates a multicore logic layer with vertically stacked Dynamic Random Access Memory (DRAM). The chief advantage of such 3D Memory Integrated architectures is the large amounts of memory that can be accessed by the on-chip logic via high bandwidth vertical interconnects. It is therefore becoming feasible to develop optimizations for Processing Near Memory architectures that enable parallel algorithms to exploit this massive memory bandwidth, potentially revolutionizing the implementation of data-intensive algorithms, including high dimension signal processing kernels. The AFRL architecture comprises the Target 3D Memory Integrated Multicore Platform for this project. While the energy cost of block memory access can be well-defined in such a platform, there is currently no method of optimizing the memory accesses of a complex algorithm and its impact on energy consumption in platforms with 3D stacked memory. In particular, the high bandwidth connection between logic and memory in 3D architectures has created an opportunity to redesign the conventional processor-memory cache hierarchy. The specific design of the cache hierarchy and its interconnection to the large on-chip buffers on the 3D memory controllers is expected to have a significant effect on both kernel performance and overall energy-efficiency. Mapping applications to this target platform is therefore complex due to these large number of architectural features and their complex performance-energy-efficiency trade-offs. In this project we developed a performance model of the Target 3D Memory Integrated Multicore Platform that can be used for evaluating the energy-efficiency and performance-energy tradeoffs of specific signal processing algorithms as mapped on to this target platform. We then demonstrated the efficacy of this framework by mapping representative signal processing kernels to the Target 3D Memory Integrated Multicore Platform and generating design curves describing the trade-off between energy efficiency and algorithm performance. Our results enable the principled and practical exploration of parallel algorithm performance and energy-efficiency and understanding of the impact of architectural design choices on performance-energy trade-offs which is central to the adoption of 3D memory centered platforms in Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance (C4ISR) applications.

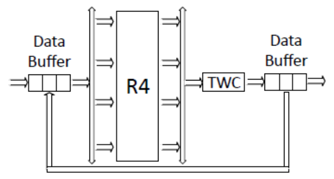
2. INTRODUCTION

Recent research has developed memory-centered architectures for organizing the 3D memory and logic layers. We denote by 3D Memory Integrated Architecture (3DMIA) a specific arrangement of logic and memory layers within a 3D Integrated Circuit (3DIC) and their interconnections. The chief advantage of such architectures is the large amounts of memory that can be accessed by the on-chip logic via high bandwidth vertical interconnects. It is therefore becoming feasible to develop optimizations for Processing Near Memory architectures that enable parallel algorithms to exploit this massive memory bandwidth, potentially revolutionizing the implementation of data-intensive algorithms, including high dimension signal processing kernels. The Air Force Research Laboratory (AFRL) is developing a secure processor that integrates a multicore logic layer with vertically stacked DRAM. This architecture represents one such 3DMIA, henceforth denoted as the Target 3D Memory Integrated Multicore Platform (Target 3D MI-MC platform). However, as compared to conventional 2D memory, access to 3D memory is complicated by its multilayered layout and energy usage patterns. While the average energy cost of memory access is lower in 3DICs owing to the shorter interconnect distribution [1], the overall energy cost becomes significant when data-intensive algorithms are executed. The challenge is therefore to develop algorithmic optimizations that enable data-intensive parallel algorithms to exploit the massive memory interconnect while meeting performance goals of throughput and latency while keeping energy consumption in check. While the energy cost of block memory access can be well defined, there is currently no method of optimizing the memory accesses of a complex algorithm and its impact on energy consumption in platforms with 3D stacked memory. Mapping applications to such platforms is complex due to the large number of architectural features and their complex performance-energy-efficiency trade-offs. In addition, the high bandwidth connection between logic and memory in 3D architectures has created an opportunity to redesign the conventional processor-memory cache hierarchy. The specific design of the cache hierarchy and its interconnection to the large on-chip buffers on the 3D memory controllers is expected to have a significant effect on both kernel performance and overall energy-efficiency.

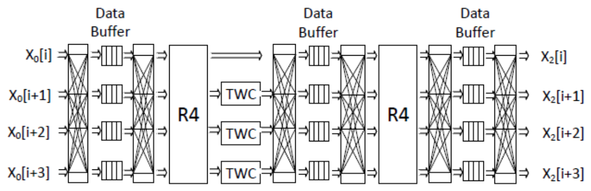
3D Integrated Circuits (3DIC) technology refers to methods being developed to stack multiple layers of logic or memory devices vertically and to connect such layers with high-bandwidth vertical interconnects. Current interconnects are based on Through Silicon Vias (TSVs) that pass through the silicon substrates of the active layers [2]. 3DIC technology is expected to reduce interconnect distances by 200× compared to PoP/package-on-package, leading to shorter wire-length distribution, with the greatest reduction in the longest paths [3-5].

Architectures: 3DIC architectures refer to the different methods of organizing processor and memory components within a 3DIC. These include *memory-on-processor* architectures [6-9] where a memory stack is integrated over a processor-like logic layer in a two-tier 3DIC. The memory stack can be embedded DRAM prototype [9] or Static Random Access Memory (SRAM) on a multicore processing layer [7, 8]. A single processor core can also be partitioned across tiers [1, 6]. 3D Stacked Logic-in-Memory combines the features of traditional LiM/ *Logic-in-Memory* architecture and 3D Stacked DRAM. Here, layers of Logic-in- Memory are interleaved between the 3D structure of DRAM layers. Thus, in addition to the benefits of 3D stacked memory, the benefits of having computation blocks close to memory can also be gained. 3DIC technology is being applied to develop the next generation of field-programmable gate arrays (FPGA) [10-12] as the high bandwidth interconnections particularly benefit FPGAs. 3DIC technology is most advanced in the design of memory chips. *3D Stacked DRAMs* [13] use a bit architecture-oriented concept. The combination of high bandwidth access to large banks of memory from logic layers makes 3DIC architectures attractive for new approaches of computing, unconstrained by the memory wall.

Energy-efficient FFT implementation: We briefly review the problem of efficiently (with respect to energy) computing the Fast Fourier Transform (FFT) of a vector of size n. The energy cost of implementing this specific architecture-algorithm pair depends on cost of the data transfers in the butterfly network. Other than the problem size n, two major parameters determine the energy consumption: Degree of horizontal parallelism (how many radix-4 stages are used in parallel), H_p, and Degree of vertical parallelism (number of parallel pipelines), V_p (Figure 1). Algorithm performance, including energy consumption, depends significantly on the parallelism in the FFT design. Figure 1 shows the energy consumption plot for an $n = 256$ BRAM-based FFT design with varying V_p and H_p parameters on Virtex II FPGA (energy costs were estimated using Xilinx XPower [14]).



(a) $H_p = 1, V_p = 1$



(b) $H_p = 2, V_p = 4$

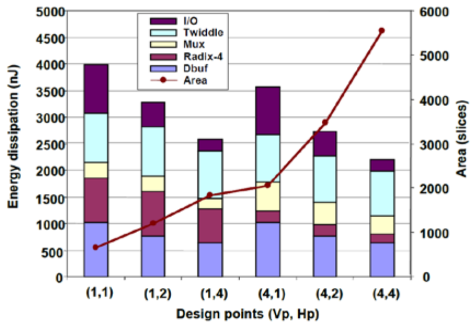


Figure 1: FFT

3. METHODS, ASSUMPTIONS, AND PROCEDURES

2D FFT can be implemented in two phases of Row FFT and Column FFT using the row-column algorithm by performing 1D FFTs on rows and columns of inputs [8]. In the Row FFT phase of the algorithm, for a problem size of N x N input matrix, 1D FFT is applied on each row of the input matrix in sequential order. The outputs of the Row FFT phase act as inputs to the Column FFT phase. In the Column FFT phase, 1D FFT is applied on each column of the N x N matrix and the outputs of Column FFT phase represent the final output of 2D FFT on the original N x N input matrix.

2D FFT on 3D memory has been the focus of many research works. In [9], memory optimized data layouts are developed for FFT on hardware accelerators such as ASIC and FPGA. Block data layout is implemented for DDR3 memory and later extended to 3D memory. A block is mapped to a row of a bank and multiple blocks are distributed among banks to increase the bandwidth of the memory. For a block of size t x t and a problem size N x N, the on-chip memory requirement is of the order $O(tN)$. In [11], a Logic-in-Memory (LiM) IC is developed to perform 2D FFT on 3D memory. Application specific logic cores are used to implement 2D FFT and energy efficiency and bandwidth are targeted as the performance metrics. Although inter-layer pipelining is utilized, block data layout from [9] is used. In [10], processing kernel on FPGA is developed to implement dynamic data layouts to reduce the number of row activations. Multiple rows/columns (p) of input data are prefetched from the memory and a permutation network is used while writing back the outputs to memory to reduce the number of row activations. The on-chip memory requirement is of the order $O(pN)$, for $1 < p < t$. None of these works focus on the on-chip memory and require substantial amount of on-chip memory to achieve high bandwidth for large problem sizes.

We developed an Optimized data layout to implement 2D FFT on 3D memory which achieves a minimum on-chip memory requirement without sacrificing the bandwidth and latency of 3D memory. We exploit inter-layer pipelining and parallel vault access to hide the latency of accessing elements in the same layer and overhead of accessing multiple rows. By achieving maximum bandwidth for both Row and Column FFT phases, our data layout stores only the necessary elements in on-chip memory and minimizes the on-chip memory requirement.

3.1 Target Architecture

Our target architecture is a 3D memory integrated FPGA consisting of 3D memory and an FFT Processing Unit (PU) on FPGA. The components of the architecture are illustrated in Figure 2.

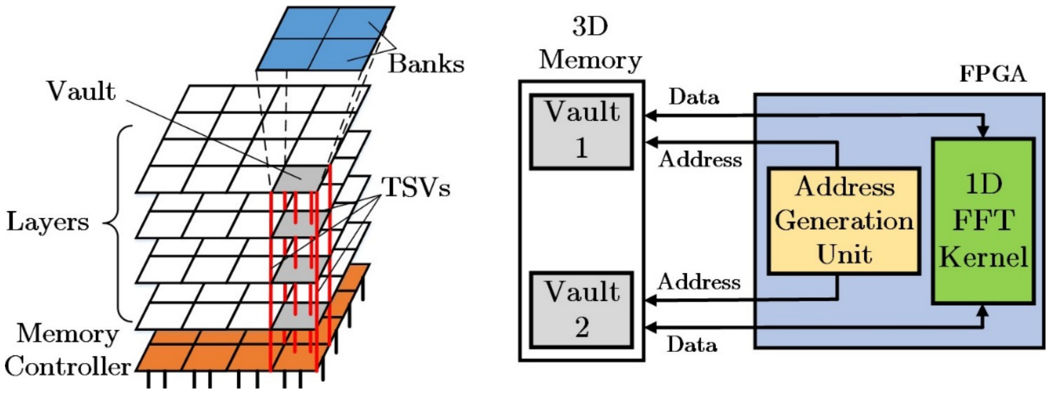


Figure 2: (a) Architecture of a 3D Memory (b) FFT PU on FPGA

3D Memory

3D memory is organized as a set of *v* vaults consisting of *l* layers and *b* banks per layer in a vault. Data in a vault is accessed using vertical interconnects (TSVs). A representative architecture of 3D memory consisting of 16 vaults with 4 layers and 4 banks per layer is illustrated in Figure 2(a). Vaults do not share TSVs with one another and hence can be accessed in parallel. Within a vault, data in different layers can be accessed at a faster rate than data in the same layer, a property known as *inter-layer pipelining* [15], [16], [17]. This is because the latency of activation overhead of rows in different layers can be overlapped due to fast TSVs. Within a layer, the structure of 3D memory is similar to the structure of DDR3 with data stored in rows and columns in each bank. Accessing data stored in different rows of the same bank incurs large latency due to row activation overhead whereas, bank interleaving can be used to reduce the latency by accessing data in different banks. Each data element stored in a 3D memory can be accessed by specifying the address in terms of vault, layer, bank, row and column. For each read/write request to the 3D memory, a specific row in a bank belonging to a layer in a vault is accessed and the bandwidth and latency of 3D memory depends on the access pattern of these requests. In our previous work [18], [19], [20], we developed a parameterized model of the 3D memory to identify the parameters which have a significant impact on the bandwidth and latency of 3D memory. Our model characterizes the 3D memory in terms of timing parameters which take into account the architecture and different access patterns. For the sake of completeness, we describe once again the parameters of the 3D memory model:

- t_{vault}: time between accesses to *different vaults*
- t_{layer}: time between accesses to *different layers* in a vault
- t_{bank}: time between accesses to *different banks* in a layer in the same vault
- t_{row}: time between accesses to *different rows* in a bank
- t_{col}: time between accesses to *different columns* in a row

FFT Processing Unit (PU) on FPGA

The FFT processing unit consists of a 1D FFT kernel and an address generation unit. 1D FFT kernel processes inputs of size *N* to produce FFT outputs. The address generation unit maps the inputs and outputs of the 1D FFT kernel to the required addresses in the memory. Since the kernel can process streaming data, we use different vaults to read inputs and write the outputs. In Figure 2, in the Row FFT phase, Vault 1 acts as the input vault and Vault 2 acts as the output vault. In the Column FFT phase, their roles are reversed.

3.2 Performance Modeling of 3D Memory Integrated Architecture

For each component of the target architecture, we describe the parameters important for performance modeling. We then map 2D FFT onto the model of the target architecture and derive the relevant equations necessary to carry out performance analysis in terms of throughput. All the parameters are defined in terms of number of processor cycles.

Data-Driven Architecture Parameterization

Figure 3 shows the 3D memory integrated architecture. 3D memory is used to store the input and output matrices of 2D FFT and consists of 2 parameters: page hit and page miss. When a memory request results in a page hit (t_{read_hit} / t_{write_hit}), the latency is 12 cycles and a page miss (t_{read_miss} / t_{write_miss}) causes a latency of 24 cycles. We assume these parameters have the same value for both read and write requests. On-chip memory can be accessed every clock cycle (t_{on_chip}). Based on empirical data, we observe that the exact computation time of the processor ($t_{compute}$) varies with the number of computations and problem size; therefore, we assume that the total computation time of a single processor for 1D FFT of "*n*" elements is

$\alpha*n$ ($0 < \alpha \leqslant 1$) and we evaluate the algorithm architecture mapping for various values of α. Data exchange across the interconnection network incurs a software overhead for each data transfer in addition to the latency of data transfer across the network. We denote these parameters as t_{mpi} and t_{xbar} respectively.

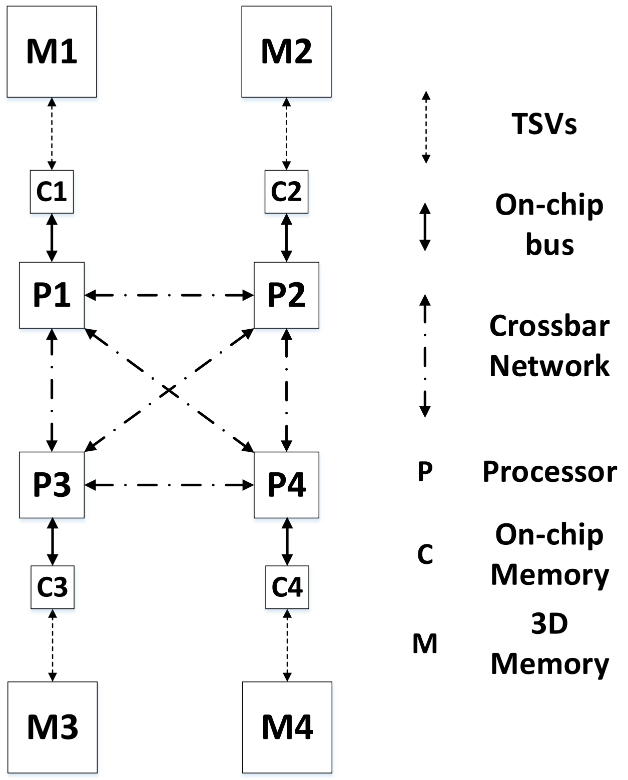


Figure 3: 3D memory integrated architecture

From the experimental values provided by AFRL, the 3D memory is a Double Data Rate (DDR) memory running at 1 Giga Hertz (GHz) with 32 Giga Bytes (GB)/s peak bandwidth. This equates to 256 bits per clock cycle (1 nanosecond (ns)). Therefore, we assume that 4 elements of 64 bits each can be read/write in 12 cycles for page hit and 24 cycles for page miss. The parameters and their values are summarized in Table 1.

Table 1: Throughput Parameters for 3D memory integrated architecture

Symbol	Parameter	Value (# cycles)
$t_{compute}$	computation time of 1D FFT for "*n*" inputs	$\alpha*n$
t_{read_hit}	read latency for page hit in the 3D memory	3
t_{read_miss}	read latency for page miss in the 3D memory	6
t_{write_hit}	write latency for page hit in the 3D memory	3
t_{write_miss}	write latency for page miss in the 3D memory	6
t_{on_chip}	access latency for read/write to on-chip memory	1
t_{xbar}	latency of data transfer ("*b*" bits) across the crossbar	$(b/64)*3$
t_{mpi}	software overhead for data transfer across the crossbar	4200*2

LogP Model

We use the LogP model [21] to develop a performance model and develop algorithm architecture mapping to arrive at the total execution time for implementing 2D FFT. Using the LogP model, we decompose the task of 2D FFT into phases consisting of local computation at each processor and communication between the processors. Parameters of LogP model are defined as:

L = an upper bound on the latency, or delay, incurred in communicating a message containing a word from its source memory module to its target memory module

o = overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message

g = gap, defined as the minimum time interval between consecutive message transmission or reception of each message

P = number of processors/memory modules

Mapping the parameters of the target architecture in Table 1 to LogP model, it can be observed that **L** = t_{xbar}; **o** = t_{mpi}; **g** = 1 and **P** = 4.

Performance Modeling Schematic

Based on the LogP model, we divide the computation of 2D FFT into 3 phases (Figure 4). Phase 1 consists of computation on each processor to perform 1D FFT on the rows of input data. Phase 2 consists of communication among processors to gather the required data for Column FFT. Phase 3 includes local computation to perform 1D FFT on the columns of input data. Figure 2 illustrates the steps involved in each of the phases. The initial algorithm-architecture mapping consists of the input $n \times n$ matrix being distributed equally among the memory of all processors. Each 3D Memory of a processor has b rows of the input matrix where, $b = n/4$. The different phases and the steps involved in these phases are explained below. Later, we propose 2 performance models depending on the when the communication takes place (Phase 2) between processors.

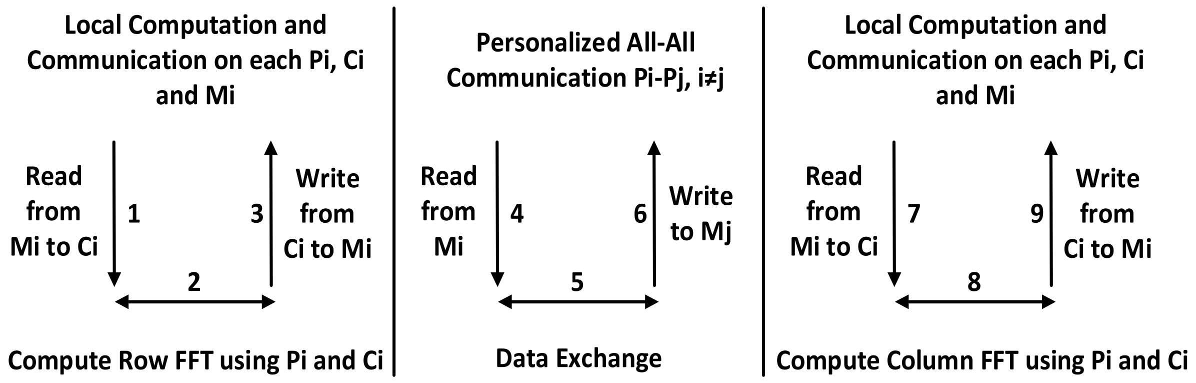

Local Computation and Communication on each Pi, Ci and Mi
Read from Mi to Ci
1
2
3
Write from Ci to Mi
Compute Row FFT using Pi and Ci
Personalized All-All Communication Pi-Pj, i≠j
Read from Mi
4
5
6
Write to Mj
Data Exchange
Local Computation and Communication on each Pi, Ci and Mi
Read from Mi to Ci
7
8
9
Write from Ci to Mi
Compute Column FFT using Pi and Ci

Figure 4: Three Phase 2D FFT Computation

3.3 Data Layouts

In this section, we describe the Baseline data layout and its limitations. Later, we present our proposed Optimized data layout along with the mapping technique. The parameters of the architecture used in our analysis and their definitions are described in Table I. We assume each access to a vault results in a column of data being available from the memory. For notation convenience, we assume there are $2v$ vaults in the memory; v vaults are used to read inputs and v vaults are used to write the outputs. This assumption does not affect our proposed data layout or the performance analysis.

Table 2: Parameters of 3D Memory

Notation	Definition
N x N	Problem size
$2v$	# vaults in 3D memory
l	# layers in a vault
b	# banks per layer in a vault
r	# rows in a bank
c	# columns in a row of a bank

Baseline Data Layout

We use the block data layout proposed in [22] as the Baseline data layout. In this data layout, a block or a tile of size t x t is mapped to a row of a bank in the memory and multiple such blocks are mapped to different banks. The value of t ranges between [1, $\sqrt{c}$]. In order to perform a 1D FFT of a row of N elements, an entire row of blocks (equivalent to t rows of N x N) are transferred from the 3D memory to on-chip memory. An FFT kernel is used to process the data stored in on-chip memory and the outputs are written back to memory. This process is repeated for all the rows in the input matrix to complete the Row FFT phase. In the subsequent Column FFT phase, entire column of blocks is transferred to the on- chip memory and processed to produce the final Column FFT outputs. Although the Baseline data layout can enable high throughput, we observe the following limitations of this data layout.

Limitation 1: Bandwidth of the 3D memory is proportional to the block size with $t = \sqrt{c}$ achieving maximum bandwidth. For $t = \sqrt{c}$, blocks are accessed from different layers and the latency overhead of accesses to the same bank is overlapped with accesses to banks in other layers and the bandwidth is limited by t_{layer}. For $t < \sqrt{c}$, the majority of the consecutive blocks are mapped to banks in the same layer and t_{bank} and t_{col} will limit the bandwidth of the 3D memory. The effect of small block sizes on performance is evident in [22], with t = (4, 8) achieving (33%, 50%) of the performance in comparison with that of t = 32.

Limitation 2: For an $N \times N$ problem size, $O(\sqrt{c}N)$ on-chip memory is required to achieve maximum bandwidth.

At any point of time, an entire row/column of blocks of data (tN elements) need to be stored in on-chip memory to process N elements of a row/column. Based on Limitation 1, maximum bandwidth is achieved for $t = \sqrt{c}$. Therefore, the on-chip memory required is $\sqrt{c}N$ elements of data. In [22], the authors use block size t = 32 to achieve maximum bandwidth. For problem sizes N = [8192, 32768] complex single-precision (2 x 32 bits per word) inputs, this translates to a large on-chip memory requirement in the range of 16-67 Mbits.

Therefore, the Baseline data layout requires large on-chip memory to achieve maximum bandwidth from 3D memory and on limited on-chip memory architectures, bandwidth of 3D memory reduces which translates to higher execution time.

Optimized Data Layout

Our data layout is defined by two mapping functions, corresponding to each phase of 2D FFT. Each function is a mapping of an N x N matrix to locations in 3D memory. A location (address) is defined by the quintuple $v(a_{ij}), l(a_{ij}), b(a_{ij}), c(a_{ij}), r(a_{ij})$ which maps matrix element a_{ij} to a vault, layer, bank, column and row in the 3D memory. The first mapping function (DL 1) describes the layout of FFT input matrix A in 3D memory before the start of the Row FFT phase. The second mapping function (DL 2) is used to write the elements of matrix A`, the output of the Row phase, to 3D memory. The same layout is then used to read columns of A` during the Column FFT phase. The outputs of this phase are the final outputs and can follow either data layout above, depending on how the resultant matrix is to be used further.

In order to derive our mapping scheme for optimal on-chip storage and bandwidth maximizing 2D FFT data layout, we make the following basic assumption about the timing parameters of 3D memory: $t_{layer} \leq \{ t_{bank}, t_{col} \} \leq t_{row}$. We also assume the number of layers is sufficient to make $l.\ t_{layer} \geq \{t_{col}, t_{bank}\}$. These assumptions are based on our estimates of the timing and architecture parameters of 3D memory, as described in [23]. The 3D memory in [23] has a peak bandwidth of 8 GB/s per vault and an element of 64 bits can be accessed for each memory request, which translates to an access time of 1 ns for each element. Therefore, we assume t_{layer} = 1 ns which represents the least possible latency of memory accesses. Further, since the structure of a layer in a 3D memory is similar to DDR3 [24], we estimate the values of other timing parameters as t_{bank} = 2 ns, t_{col} = 4 ns and t_{row} = 40 ns based on timing parameters described in [24]. The key characteristics of our mapping scheme based on the above assumptions are as follows: Since vaults can be accessed in parallel, it is trivial to distribute elements across vaults to maximize bandwidth. Our data layout further maps accesses within a vault to different layers to ensure the minimum possible latency for t_{layer}, for each access. Now, considering accesses within a vault, our layout maximizes bandwidth by hiding the latency of consecutive accesses to the same row or different rows in a bank through a number of intermediate accesses to other layers, utilizing $p.t_{layer} \geq t_{col}$ and $q.t_{layer} \geq t_{row}$. For example, choosing $p \geq 4$ and $q \geq 40$ based on the parameters above, will hide the latency of t_{col} and t_{row} and incur a minimum latency of t_{layer}. Hiding the latency of accesses to different rows and columns is possible due to the large number of banks [23] and faster access across the 3rd dimension of 3D memory [15], [17].

For notational simplicity and without loss of generality in our description of the mapping schemes, we assume that parameters N, v, l, b, r and c are powers of 2 and that $k = \sqrt{vlbc}$ is an integer (power of 2). These assumptions can be relaxed at the cost of increased notational complexity in the description of our layout scheme. We also assume $N \leq \sqrt{vlbrc}$ problem fits in memory).

Data Layout 1 (DL 1): Our first mapping scheme for the Row FFT phase is a straightforward round-robin mapping of the rows of A over vaults, layers, banks, columns and rows (Figure 5). Each row of N input elements from A is distributed in a round robin fashion across v vaults (line 3). Similarly, in a round- robin fashion, the N/v elements within a vault are distributed among l layers in that vault and the $N/(vl)$ elements within a layer distributed among its b banks (line 4). Finally, the $N/(vlb)$ elements assigned to a bank are distributed in row major order among its c columns and r rows (line 5). This mapping function is repeated for all the N rows of the input matrix.

DL 1: Mapping Function for Matrix A (Row FFT Inputs)

```
$a_{ij} : (i,j)^{th}$ element of $A$, $0 \leq i,j \leq N-1$
Address$[a_{ij}] \rightarrow \{v(a_{ij}), l(a_{ij}), b(a_{ij}), c(a_{ij}), r(a_{ij})\}$
$v(a_{ij}) = (i \cdot N + j) \bmod v$
$l(a_{ij}) = \left\lfloor \frac{(i \cdot N + j)}{v} \right\rfloor \bmod l$ ;     $b(a_{ij}) = \left\lfloor \frac{(i \cdot N + j)}{vl} \right\rfloor \bmod b$
$c(a_{ij}) = \left\lfloor \frac{(i \cdot N + j)}{vlb} \right\rfloor \bmod c$;     $r(a_{ij}) = \left\lfloor \frac{(i \cdot N + j)}{vlbc} \right\rfloor \bmod r$
```

Figure 5: DL1 Mapping Function

Data Layout 2 (DL 2): Our second mapping scheme ensures that consecutive accesses to 3D memory components (vaults, layers etc.) are sufficiently spaced to absorb respective component activation overheads both during the row major write phase at the end of the Row FFT as well as during the column major read phase at the start of Column FFT (Figure 6). Consider the same row index across all banks, layers and vaults of 3D memory. Given *c* columns per row, there are *vlbc* locations corresponding to this row index across the entire 3D memory. We want to repeatedly distribute elements from the rows and columns of the *N* x *N* output matrix *A* of the Row FFT phase uniformly among these *vlbc* locations for each row index. Note that *A* is only available one row at a time and the writing to memory occurs as per our mapping function after each row of *A* becomes available. We start by dividing *A* into contiguous *k* x *k* blocks, with *k* = *vlbc*. It should be note that although we divide the matrix into blocks, our blocks as well as our mapping function are quite different from the Baseline data layout. DL 2 describes in detail each of the mapping functions.

DL 2: Mapping Function for matrix $\widehat{A}$ (Row FFT Outputs)

```
$a_{ij} : (i,j)^{th}$ element of $\widehat{A}$, $0 \leq i,j \leq N-1$
Address$[a_{ij}] \rightarrow \{v(a_{ij}), l(a_{ij}), b(a_{ij}), c(a_{ij}), r(a_{ij})\}$
$\{k, y\}$ // block related parameters
$v(a_{ij}) = (i + j) \bmod v$
$l(a_{ij}) = (\lfloor \frac{i}{v} \rfloor + \lfloor \frac{j}{v} \rfloor) \bmod l$
$b(a_{ij}) = (\left\lfloor \frac{i}{vly} \right\rfloor + \left\lfloor \frac{j}{vly} \right\rfloor) \bmod b$
$r(a_{ij}) = \frac{N}{k} \lfloor \frac{i}{k} \rfloor + \lfloor \frac{j}{k} \rfloor$
```

Figure 6: DL2 Mapping Function

4. RESULTS AND DISCUSSION

4.1 Performance Evaluation

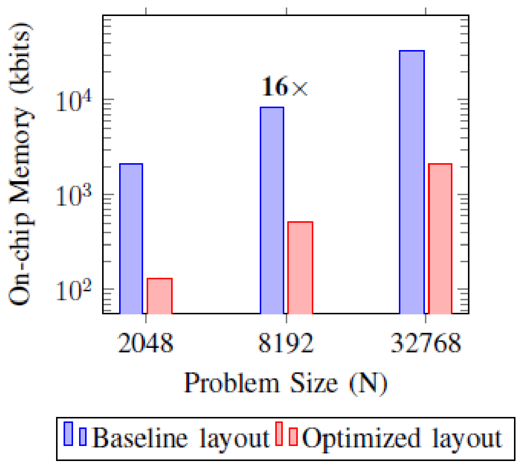
3D memories have become popular recently, and since the exact internal architecture is proprietary, existing cycle accurate simulators do not capture all the features of the 3D memory. For example, [25], [26] do not provide the feature of inter-layer pipelining and are limited to specific types of 3D memory. We do not claim cycle accurate performance comparison as we are looking for higher order performance estimate of 2D FFT on 3D memory.

For the performance analysis, timing parameters of 3D memory are estimated as: t_{layer} = 1 ns, t_{bank} = 2 ns, t_{col} = 4 ns and t_{row} = 40 ns [23], [24]. We assume vaults can be accessed in parallel making v elements available from v vaults in a time equal to the latency of accessing one element from 1 vault, i.e., $t_{vault} = 0\ ns$. We assume the inputs are complex single-precision floating point numbers (2 32 bits per word) and each access to a vault ensures 1 column/element of data, i.e., 64 bits are available to the FPGA. The parameters of 3D memory are tabulated in Table III. We assume a streaming FFT Processing Unit on FPGA with 128 Gbits/s (16 GB/s) throughput. For a vault with a bandwidth of 8 GB/s [23], 2 vaults saturate the throughput of the FFT processing unit. Therefore, 2 vaults are used to read inputs and 2 vaults are used to store the outputs.

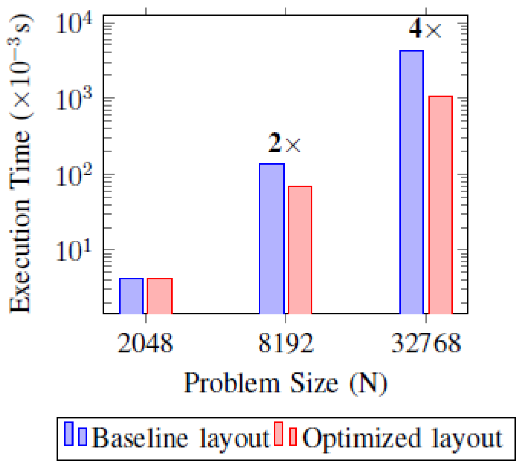
Table 3: 3D Memory Parameter Values

Parameter	v	l	b	r	c	Vault Bandwidth
Values	4	4	4	4096	256	8 GB/s

In Figure 7(a), we analyze the amount of on-chip memory required to achieve maximum bandwidth for Baseline and Optimized data layouts. The Baseline data layout uses a block size of t = 16 and on-chip memory of 33 Mbits to achieve maximum bandwidth. We observe that the Optimized data layout achieves maximum bandwidth with substantially lower on-chip memory (16x). In Figure 2(b), we assume the architecture has a limited on-chip memory of 4 Mbits. For the Baseline data layout, the available on-chip memory is sufficient to achieve maximum bandwidth for small problem sizes (N = 2048). For large problem sizes (N = 8192, 32768), due to small amount of on-chip memory, Baseline data layout is restricted to small block sizes and the majority of the consecutive accesses are mapped to the same layer and the bandwidth is limited by t_{bank} or t_{col}. This translates to a higher execution time in comparison with the Optimized data layout. On the other hand, the Optimized data layout does not suffer any degradation in performance since the available on-chip memory is sufficient to store the required O(N) elements of input data and ensures maximum bandwidth is achieved resulting in 2× to 4× reduction in execution time.



(a) On-chip Memory for Maximum Bandwidth



(b) Execution Time on Limited On-chip Memory

Figure 7: Evaluation of Optimized Layouts

4.2 Performance Analysis

For the sake of completeness, we reiterate the 3 phases in the computation of 2D FFT on the target architecture.

- Phase 1: Each processor computes 1D FFT on the rows of input data present in its private 3D memory.
- Phase 2: Processors communicate among each other to gather the results of Row FFT which act as input data for Column FFT.
- Phase 3: Local computation to perform 1D FFT on the columns of input data.

We also briefly describe the 2 models we developed in the previous quarter to accommodate different interconnection networks.

Model 1: In this model, outputs of Row FFT are exchanged among processors after each row is processed. This is due to the limitation of interconnection network in transferring large block sizes.

Model 2: If the interconnection network allows a sufficiently large block size, data exchange happens after Row FFT computation of all the rows have been completed. This results in minimum overhead due to communication between processors.

Equations are tabulated below in Table 4.

Table 4: Equations for Performance Models

Model 1	Model 2
$t_{phase1} = n*t_{read} + t_{compute} + n*t_{write}$	$t_{phase1} = n*t_{read} + t_{compute} + n*t_{write}$
$t_{phase2} = [b*t_{read} + t_{mpi} + t_{xbar} + b*t_{write}]*3$	$t_{phase2} = [b^2(t_{read}) + t_{mpi} + t_{xbar} + b^2(t_{write})]*3$
$t_{phase3} = n*t_{read\_miss} + t_{compute} + n*t_{write}$	$t_{phase3} = n*t_{read\_miss} + t_{compute} + n*t_{write}$
Total Execution Time = $[t_{phase1} + t_{phase2} + t_{phase3}](n/4)$	Total Execution Time = $t_{phase1}*(n/4) + t_{phase2} + t_{phase3}*(n/4)$

We present the performance of both the models for different problem sizes in Table 5. We also present the effect of variation in α on performance.

Table 5: Performance Comparison: Model 1 and Model 2

$n \times n$	α	Performance (GOPS)		Improvement
		Model 1	Model 2	Model 2 vs Model 1
1024 × 1024	0.5	3.59	7.44	2.07×
	1	3.52	7.13	2.03×
4096 × 4096	0.5	7.06	8.96	1.27×
	1	6.82	8.59	1.26×
8192 × 8192	0.5	8.56	9.71	1.14×
	1	8.24	9.3	1.13×

For small problem sizes, Model 2 achieves a significant improvement in performance compared with Model 1. On the other hand, as the problem size increases, the read and write latency to memory dominates the total execution time and the difference in performance between the 2 models is marginal.

We also evaluate the following optimizations to reduce the communication overhead and achieve higher throughput for 2D FFT implementation on 3D memory integrated architectures.

Dynamic Data Layout: While performing data exchange between the processors, we write the data in transpose format to the destination memory so that the future accesses for column FFT will result in a sequential row accesses. For example, read the data sequentially from the memory of Processor 2 and while writing this data to the memory of Processor 1, we write the data into different rows. Although we incur a penalty while writing, the reduction in latency while performing column FFT makes up for this penalty.

Overlapping Communication and Computation: In an ideal scenario, the exchange of data between processors should not be visible to the processors so that the processors are not idle. This results in peak performance. To achieve the peak performance, we need to minimize the data exchange overhead by overlapping computation time with data exchange time.

Data exchange between processors can be done at various levels of granularity. The two extremes are:
(a) Data exchange after every row FFT is finished
(b) Data exchange after the entire set of rows ($n/4$) FFT is finished

We have observed that if the data exchange is done after every row, the overhead is too high and very large compared to local computation phase. On the other hand, if data exchange is done after all $n/4$ rows FFT are finished; the local computation phase is much larger in magnitude. Therefore, there exists an optimum granularity, i.e., after an "x" number of rows FFT have finished; data exchange for those rows can be performed.

Equating the total computation time for "x" rows with the data exchange time for "x" rows gives us:
$(n*t_{read} + \alpha*n + n*t_{write})*x = [(n/4)\,(t_{read})\,*x + \mathbf{4200*2} + (n/4)(3)*x + (n/4)(t_{write})*x]*3$

By solving the above equations using the values from Table 4, we obtain: $x = 8400*3/(n(\alpha - 0.75))$

Note: this is only applicable when $\alpha > 0.75$.

Therefore, we can start the data exchange phase after "x" rows FFT has been computed. Hence, the total overhead of data exchange phase is equal to the initial latency to compute "x" rows FFT. Using the above value of "x" in Model 2 we obtain the total execution time:
Total time = $3.75*n^2 + \alpha*(n^2/2) + 8400*3*(6 + \alpha)/(\alpha - 0.75)$

With optimization #1 enabled, we observe that minimum value of α increases to 3. So, $x = 8400*3/(n(\alpha - 3))$
An analytical performance comparison of baseline and optimized data layout is shown in Table 6.

Table 6: Performance Comparison of Baseline vs Optimized Data Layout

Baseline implementation:	**Optimized implementation:**
$t_{phase1} = n*t_{read} + \alpha*n + n*t_{write}$	$t_{phase1} = n*t_{read} + \alpha*n + n*t_{write}$
$t_{phase2} = (n^2/16)(t_{read}) + \mathbf{4200*2} + (n^2/16)(3) + (n^2/16)(t_{write})$	$t_{phase2} = (n^2/16)(t_{read}) + \mathbf{4200*2} + (n^2/16)(3) + (n^2/16)(t_{write\_miss})$
$t_{phase3} = n*t_{read\_miss} + \alpha*n + n*t_{write}$	$t_{phase3} = n(t_{read}) + \alpha*n + n(t_{write})$
Total Time = $5.4375*n^2 + \alpha*(n^2/2) + 8400*3$	Total Time = $5.25*n^2 + \alpha*(n^2/2) + 8400*3$

Empirical Performance Evaluation
Below, we present the performance of various models for different problem sizes. We also present the effect of variation in α on performance of various models.
$\alpha = 0.5$

Model #	Performance in GOPS		
	1024 × 1024	4096 × 4096	8192 × 8192
1	7.44	8.96	9.71
2	3.59	7.06	8.56
3	0.027	0.032	0.035

$\alpha = 1$

Model #	Performance in GOPS		
	1024 × 1024	4096 × 4096	8192 × 8192
1	7.13	8.59	9.3
2	3.52	6.82	8.24
3	0.027	0.032	0.035

Performance Improvement using Optimizations
$\alpha = 1$

Model #	Optimization #	Performance in GOPS		
		1024 × 1024	4096 × 4096	8192 × 8192
1	None	7.13	8.59	9.3
1	1	7.36	8.87	9.61
2	2	8.63	11.88	12.97

5. CONCLUSIONS
While the energy cost of block memory access can be well-defined, there is currently no method of optimizing the memory accesses of a complex algorithm and its impact on energy consumption in platforms

with 3D stacked memory. Mapping applications to such platforms is complex due to the large number of architectural features and their complex performance-energy-efficiency trade-offs. In addition, the high bandwidth connection between logic and memory in 3D architectures has created an opportunity to redesign the conventional processor-memory cache hierarchy. The specific design of the cache hierarchy and its interconnection to the large on-chip buffers on the 3D memory controllers is expected to have a significant effect on both kernel performance and overall energy-efficiency.

In this effort, we first developed a model-based optimization framework for implementing signal processing algorithms in an energy-efficient manner on a 3D memory-integrated multicore architecture. We demonstrated the efficacy of this framework by mapping representative signal processing kernels to the Target 3D MI-MC platform. Specifically, our performance modeling approach was developed to enable the following capabilities.

- Quantify the expected performance metrics of throughput, latency, and energy-efficiency of a given algorithm after it is mapped on to the performance model.
- Evaluate the impact of alternate designs of the placement of the processor-memory cache on the performance of an algorithm.
- Evaluate the impact of accessing 3D memory buffers and interfaces via an interposer layer.
- Quantify the expected trade-off between performance and energy-efficiency of the target platform when a specific application kernel is executed on it.

We also developed an on-chip memory efficient data layout to implement 2D FFT on 3D memory. Our data layout exploits inter-layer pipelining and parallel vault access to hide the latency overhead of strided accesses. The data layout ensures maximum bandwidth is available from 3D memory with the on-chip memory requirement of $O(\sqrt{N})$ for a problem size of N x N. In comparison with the Baseline data layout, our Optimized data layout reduces the on-chip memory by $\sqrt{c}$ for c columns in a row of a memory bank. With limited on- chip memory, our data layout achieves 2 to 4 reduction in execution time compared with the Baseline data layout.

6. REFERENCES

[1] B. Black, D. W. Nelson, C. Webb, and N. Samra, "3D processing technology and its impact on iA32 microprocessors," in Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on, 2004, pp. 316-318.

[2] E. Beyne, "Through-Silicon via Technology for 3D IC," in Ultra-thin Chip Technology and Applications, ed: Springer, 2011, pp. 93-108.

[3] S. Das, A. Fan, K.-N. Chen, C. S. Tan, N. Checka, and R. Reif, "Technology, performance, and computer- aided design of three-dimensional integrated circuits," in Proceedings of the 2004 international symposium on Physical design, 2004, pp. 108-115.

[4] A. Rahman and R. Reif, "Thermal analysis of three-dimensional (3-D) integrated circuits (ICs)," in Interconnect Technology Conference, 2001. Proceedings of the IEEE 2001 International, 2001, pp. 157- 159.

[5] S. Das, A. Chandrakasan, and R. Reif, "Design tools for 3-D integrated circuits," in Proceedings of the 2003 Asia and South Pacific Design Automation Conference, 2003, pp. 53-56.

[6] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, and D. Pantuso, "Die stacking (3D) microarchitecture," in Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on, 2006, pp. 469-479.

[7] S. K. Lim, "3D-MAPS: 3D massively parallel processor with stacked memory," in Design for High Performance, Low Power, and Reliable 3D Integrated Circuits, ed: Springer, 2013, pp. 537-560.

[8] D. Fick, R. G. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, and N. Liu, "Centip3De: A Cluster-Based NTC Architecture With 64 ARM Cortex-M3 Cores in 3D Stacked 130 nm CMOS," Solid-State Circuits, IEEE Journal of, vol. 48, pp. 104-117, 2013.

[9] M. Wordeman, J. Silberman, G. Maier, and M. Scheuermann, "A 3D system prototype of an eDRAM cache stacked over processor-like logic using through-silicon vias," in Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International, 2012, pp. 186-187.

[10] M. Leeser, W. M. Meleis, M. M. Vai, S. Chiricescu, W. Xu, and P. M. Zavracky, "Rothko: A three-dimensional FPGA," Design & Test of Computers, IEEE, vol. 15, pp. 16-23, 1998.

[11] A. Gayasen, V. Narayanan, M. Kandemir, and A. Rahman, "Designing a 3-D FPGA: switch box architecture and thermal issues," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 16, pp. 882-893, 2008.

[12] C. Ababei, P. Maidee, and K. Bazargan, "Exploring potential benefits of 3D FPGA integration," in Field programmable logic and application, ed: Springer, 2004, pp. 874-880.

[13] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in ACM SIGARCH Computer Architecture News, 2008, pp. 453-464.

[14] (2014-01-14). Xilinx XPower Analyzer. Available: http://www.xilinx.com/products/design_tools/logic_design/verification/xpower_an.htm

[15] Qiuling Zhu, Bilal Akin, H Ekin Sumbul, Fazle Sadi, James C Hoe, Larry Pileggi, and Franz Franchetti. A 3D-Stacked Logic-in-Memory Accelerator for Application-Specific Data Intensive Computing. In 3D Systems Integration Conference (3DIC), 2013 IEEE International, pages 1–7. IEEE, 2013.

[16] Donghyuk Lee, Saugata Ghose, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Simultaneous Multi-Layer Access: Improving 3D- Stacked Memory Bandwidth at Low Cost. ACM Trans. Archit. Code Optim., 12(4):63:1–63:29, January 2016.

[17] Feihui Li, Chrysostomos Nicopoulos, Thomas Richardson, Yuan Xie, Vijaykrishnan Narayanan, and Mahmut Kandemir. Design and Management of 3D Chip Multiprocessors using Network-in-Memory. ACM SIGARCH Computer Architecture News, 34(2):130–141, 2006.

[18] Shreyas G Singapura, Anand Panangadan, and Viktor K Prasanna. Towards Performance Modeling of 3D Memory Integrated FPGA Architectures. In Applied Reconfigurable Computing, pages 443–450. Springer, 2015.
[19] Shreyas G Singapura, Anand Panangadan, and Viktor K Prasanna. Performance Modeling of Matrix Multiplication on 3D Memory Integrated FPGA. In Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International, pages 154–162. IEEE, 2015.
[20] Ren Chen, Shreyas G Singapura, and Viktor K Prasanna. Optimal Dynamic Data Layouts for 2D FFT on 3D Memory Integrated FPGA. In Parallel Computing Technologies, pages 338–348. Springer, 2015.
[21] Culler, David, et al. LogP: Towards a realistic model of parallel computation. Vol. 28. No. 7. ACM, 1993.
[22] Berkin Akin, Franz Franchetti, and James C Hoe. Understanding the Design Space of Dram-Optimized Hardware FFT Accelerators. In Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on, pages 248–255. IEEE, 2014.
[23] Micron HMC. http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf.
[24] Micron DDR3 Datasheet. https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr3/2gb_ddr3_sdram.pdf.
[25] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. IEEE Computer Architecture Letters, PP(99):1–1, 2015.
[26] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. CACTI-3DD: Architecture-Level Modeling for 3D Die-Stacked DRAM Main Memory. In Proceedings of the Conference on Design, Automation and Test in Europe, pages 33–38. EDA Consortium, 2012.

APPENDIX A - PUBLICATIONS

A-1. Kartik Lakhotia, Shreyas Singapura, Rajgopal Kannan, Viktor Prasanna, ReCALL: Reordered Cache Aware Locality based Graph Processing, IEEE International Conference on High Performance Computing, Data and Analytics, pp. 1-10, October 2017.

2017 IEEE 24th International Conference on High Performance Computing (HiPC)

ReCALL: Reordered Cache Aware Locality based Graph Processing

Kartik Lakhotia*, Shreyas Singapura*, Rajgopal Kannan†, Viktor Prasanna*
*Ming Hsieh Department of Electrical Engineering,
University of Southern California
{klakhoti, singapur, prasanna}@usc.edu
†US Army Research Lab
rajgopal.kannan.civ@mail.Mil

***Abstract*—Sparse graph processing generates highly irregular Memory Access Patterns (MAP) which lack locality and result in poor cache performance. In this paper, we propose a novel graph ordering algorithm that addresses this problem. We observe that existing reordering algorithms primarily try to improve cache line utilization by enhancing spatial locality. They are oblivious to cache data reuse which reflects the temporal locality that MAP can possess. Our premise is that peak efficiency can be achieved by a graph order for which the resulting MAP exhibit both spatial and temporal locality. Therefore, we first introduce a new metric *Profit*, that quantifies cache data reuse leading to a heuristic *pH* that enhances temporal locality in the MAP of graph algorithms. Then we define a notion of dynamically matching MAP with cache contents in a way that jointly maximizes both cache data reuse and cache line utilization. To perform this joint optimization, we develop a *Block Reordering* algorithm which utilizes *pH* to rearrange blocks of consecutive nodes with high spatial locality. We evaluate our algorithm using 8 real world datasets and 4 representative graph algorithms. Experimental results show that graphs obtained by *Block Reordering* can achieve upto 2.3× speedup over the original graph order and consistently outperform the existing state of the art reordering technique by 20% to 25% reduction in cache misses.**

***Index Terms*—Graph Analytics; Big Data; Cache Performance; Data Layout; Graph Reordering**

I. Introduction

Large scale graph analytics has become increasingly important in the era of big data. Many real world applications like social networks, world wide web, road connections are represented as graphs [1]. The rapid growth in size of these graphs has led to development of various graph processing frameworks in recent years. Although the main memory capacity has grown to be able to fit even large graphs on a single server, it still remains challenging to efficiently utilize the computing resources on conventional systems. This is because of the high communication to computation ratio of graph algorithms and poor locality in their access patterns [2], [3]. As shown in [4], a large fraction of execution time is wasted on DRAM access latency. CPUs rely heavily on caches for high bandwidth and low latency data access but due to the irregular Memory Access Pattern (MAP) of graph algorithms, loads and stores often incur cache misses causing processor to stall. The focus of this paper is to improve cache performance for efficient graph processing.

Caches communicate with main memory in quantum of cache lines. In order to utilize cache lines efficiently, it is desirable for the MAP to have high spatial locality. Further, cached data is efficiently reused if the MAP has high temporal locality. Therefore, increasing locality is crucial to improving cache performance of graph algorithms.

In this paper, we use node reordering to increase cache hit ratio for a large class of graph algorithms. The fundamental concept behind reordering is to alter the arrangement of node data in main memory in a way that increases locality in the MAP [5], [6], [7], [8]. In [4] and [9], reindexing is used to increase cache line utilization for graph processing algorithms. In this paper, we propose a novel reordering algorithm that improves cahce data reuse along with cache line utilization. Our algorithm dynamically matches access patterns with cache contents to enhance both spatial and temporal locality in the MAP. The major contributions of our work are:

- A method to quantify temporal locality in MAP based on amortized cache dependencies between graph nodes, which leads to a heuristic that increases reuse of cached data. We denote the quantifying metric *Profit* and the derived **cache aware** heuristic ***pH***.
- A novel *Block Reordering* algorithm that performs **combined optimization** over spatial and temporal locality by reordering blocks of nodes in the same cache line. To the best of our knowledge, this is the first work that comprehensively addresses the issue of improving cache performance using node reordering.

We evaluate our reordering using 4 representative graph algorithms on large scale real world datasets and demonstrate 1.3× to 2.3× speedup over original and optimized graph orders [9]. Experimental results show that the *Block Reordering* algorithm achieves 20% to 25% reduction in cache misses compared to existing state of the art [4].

Rest of the paper is organized as follows: Section II provides motivation and related work on cache optimization for graph processing; Section III explains the *Profit* metric that captures

This material is based on work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086, National Science Foundation (NSF) under Contract Number CNS-1643351 and ACI-1339756 and Air Force Research Laboratory under Grant Number FA8750-15-1-0185. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or NSF. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

temporal locality and defines a *Profit* maximization problem based on reordering; Section IV describes a greedy solution to *Profit* maximization and the *Block Reordering* algorithm; Section V discusses the complexity of our reordering algorithm and the optimized data structures used to implement it; and lastly, Section VI highlights the performance comparison of our reordering with state of the art techniques and Section VII concludes the paper.

II. BACKGROUND

A. Motivation

Graph Representation: We consider the adjacency matrix of graph to be stored in Compressed Sparse Row (CSR) format for the purpose of illustration. CSR representation is equivalent to adjacency list with all adjacencies densely packed in a single array. CSR is a standard format used in high performance applications [10]. It stores the graph in two arrays: vertex array (*VA[]*) and edge array (*EA[]*). The edges are sorted by destination and the source node labels of all edges are stored in *EA[]*. *VA[]* stores offsets into *EA[]* providing the location of first incoming edge towards each vertex. Additional arrays *attr[]* and *weight[]* can be used to store attributes of graph vertices and the edge weights, resepectively. We define a dataset as large graph if number of vertices are much larger than the cache capacity. Consequently, neither of the *VA[]*, *EA[]* and *attr[]* can fit in cache and must be read from main memory.

Performance Bottleneck: In this paper, we focus on a broad category of graph alorithms which are Stationary [11]. These algorithms have a property that in each iteration, all vertices are active and process their edges. Algorithm 1 depicts a kernel that is common to all stationary algorithms. SSSP

Algorithm 1 Kernel

```
1: for all nodes $v \in V$ do
2:     for all edges $(u, v) \in E$ do
3:         $attr[v] = f(attr[u], attr[v], weight(edge))$
```

using Bellman-Ford, Label Propagation, Louvain modularity, algorithms that use SpMV kernels like Pagerank etc. are examples of some algorithms that fall under this category. For such algorithms, CSR supports regular reads from *VA[]* and *EA[]* which enables the existing memory architectures to deliver high streaming performance. However, accesses *attr[u]* are indexed by the source node labels in *EA[]*. Fig. 1 shows that this access pattern can be highly irregular and spread randomly across the array. As a result, requests to source node attributes often incur main memory access latency, which becomes the bottleneck in graph processing [12]. For the rest of the paper, we'll consider accesses only to *attr[]* while examining memory performance of graph algorithms.

B. Related Work

Considering the challenges mentioned in Section II-A, cache performance and access pattern locality for graph analytics has garnered lot of interest in the scientific community. This

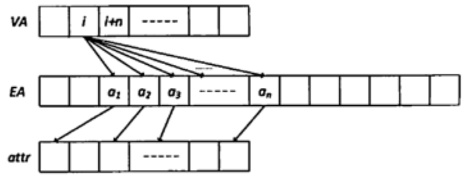


Fig. 1: While reading adjacencies of a vertex accesses consecutive locations in *EA[]*, accesses to *attr[]* are data dependent and highly irregular.

problem has been approached from different perspectives. Researchers have tried to enhance data locality by targetting algorithmic modifications: in [13], tiling and efficient data layouts are used to improve performance of Floyd-Warshall. [14], [15], [16] split PageRank and SpMV computation into phases to produce cache friendly access patterns. However these are algorithm specific optimizations that are employed at runtime.

In this paper, we specifically target the problem of reordering the graph vertices to improve cache performance for a large class of algorithms. Reordering has been used in the past to increase cache hit rate - RCM [17] and Depth first search [18] use graph traversal to create clusters of nodes in same tree levels; METIS [19] partitions the graph into clusters such that highly connected nodes lie in same cluster. However, a recent study [4] showed that tree based and clustering strategies do not scale well with size and complexity of graphs because of the absence of good edge-cuts in power law graphs and small partition sizes required to represent cache lines. In [4], a new algorithm GOrder is proposed that places the nodes that are frequently accessed together, closely in the graph ordering to improve spatial locality of references. It overcomes the drawbacks of partitioning by using a sliding window model to compute relationship of a node v with other nodes in the new graph order. Another reordering based approach that uses frequency based clustering (FC) to improve utilization of each cache line was recently proposed in [9]. FC performs an intelligent degree sort that maintains the locality in original graph order. In [9], it is shown that FC can reduce cache contention by packing high degree nodes closer in the memory.

We observe that the recent works have focussed on improving utilization of data in cache line by increasing spatial locality of data references. Even by optimizing just spatial locality, the GOrder algorithm achieves significant speedup over skewing and cluster partitioning methods, making it the most cutting-edge reordering amongst the works published to date. However, cache hit rate can also be improved by reusing the cached data which is directly related to temporal locality in data access patterns. Note that the two types of locality are orthogonal to each other and optimizing one can leave the other unaddressed. In this paper, we propose a novel algorithm that comprehensively targets both factors behind cache performance. By performing joint optimization over spatial and temporal locality, our reordering is able to outperform existing state of the art algorithms. In [20], the benefits of a vertex scheduling that matches access pattern with cache contents are explored. Such a scheduling would optimize both temporal and spatial locality but [20] does not propose any algorithm to achieve it.

As temporal locality in graph algorithms has not been formally addressed in the previous works, we first develop a model and a quantification method for it, which is discussed in the next section.

III. Problem Formulation

Given a directed graph *G(V, E)*, for any node $u \in V$, we define its in-neighbor set $N_{in}(u)$ as $\{v \in V \mid (v,u) \in E\}$. Similarly, a set of out-neighbors $N_{op}(u)$ is defined as $\{v \in V \mid (u,v) \in E\}$. Consider the CSR format explained in Section II-A. Every node of the graph is associated with an index that represents address of its attribute in the *attr[]* array. Any reference to node i can lead to two possibilities :

- If *attr[i]* is present in cache, the request will be served by cache.
- If *attr[i]* is not present in cache, it will be fetched from DRAM and a copy will be stored in cache for future use.

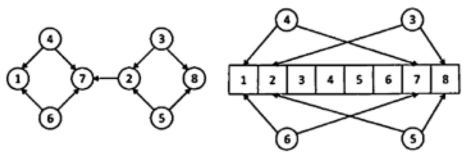
Cache capacity in typical systems is limited and cannot accommodate the entire *attr[]* array for large graphs. Therefore, node attributes get evicted and brought into the cache multiple times over the course of an algorithm. This sequence of eviction and main memory fetch is determined by the memory access pattern of the algorithm. An access pattern that references same set of nodes within a close time interval will exhibit high temporal locality and higher cache hit ratio. To study the memory access patterns, we examine the kernel given in Algorithm 1 that is central to all the stationary algorithms.

As discussed previously, the primary bottleneck limiting the performance of graph algorithms is the access to *attr[]* array. We observe that the memory references made by the kernel shown in Algorithm 1 *are completely determined by the source node labels in EA[]*. Therefore, the task of increasing temporal locality in graph processing reduces to finding a *permutation* that results in local regions of *EA[]* having identical sets of values. Moreover, for a given *EA[]*, we have a priori knowledge of the access pattern of such graph algorithms. We use this information to derive a novel metric *Profit* that quantifies reuse of cached data for a given graph.

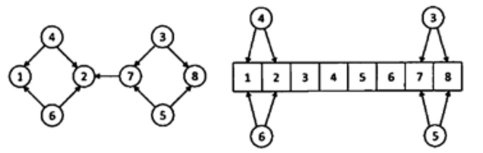
A. *PROFIT*

When a graph algorithm processes vertex i, all the nodes $v \in N_{in}(i)$ are read and brought into the cache. If the algorithm next processes node j, it will read $v \in N_{in}(j)$. Some of these reads will be cache hits if they reference the same nodes that were accessed while processing i. We define the *Profit* earned by processing j after i as the number of identical neighbors accessed and thus, the resulting cache hits achieved. Mathematically, the *Profit* relationship is defined as:

$$\mathit{Profit}(i,j) = N_{in}(i) \cap N_{in}(j)$$



(a) Random Labeling



(b) *Profit* maximized Π_{opt} Labeling

Fig. 2: Graph with different node orders (left) and corresponding data layout in memory (right). Π_{opt} increases locality in memory accesses.

While computing the net profit of a graph $P_{net}(G)$, we consider the fact that nodes are processed in the order of their indices. Therefore, $P_{net}(G)$ is a running accumulation of the *Profit* of successive nodes given by:

$$P_{net}(G) = \sum_{i=1}^{|V|-1} \mathit{Profit}(i, i+1) = \sum_{i=1}^{|V|-1} N_{in}(i+1) \cap N_{in}(i) \tag{1}$$

A node order that achieves the maximum *Profit* for a given graph structure would exhibit high temporal locality. Let Π be an arbitrary vertex labeling of V. Given a directed graph G, we define *Profit* maximization $\mathcal{P}(G)$ as the problem of finding the optimum mapping Π_{opt}, that achieves highest possible $P_{net}(G)$. From Eq. 1, we get:

$$\mathcal{P}(G) = \arg\max_{\Pi} \sum_{i=1}^{|V|-1} N_{in}(\Pi(i+1)) \cap N_{in}(\Pi(i)) \tag{2}$$

Fig. 2 shows how Π_{opt} can improve temporal locality in a graph. Fig. 2a shows a graph G with vertices labeled randomly and its layout in the main memory. Let the system processing G have a hypothetical cache with capacity equal to 2 nodes. To process G with node order given in fig. 2a, it will take 8 fetches from main memory because none of the nodes reuses the data in cache brought by its predecessor. Compare it with the same graph but with vertices labeled by Π_{opt} as shown in fig. 2b. On the same system, processing the graph in new order will only take 4 fetches from the main memory while 4 of the loads issued by CPU will be cache hits.

Lemma 1. *Obtaining optimum solution of $\mathcal{P}(G)$ is NP-Hard.*

Proof. Let $G(V,E)$ be an arbitrary graph on which we wish to find a Hamiltonian Path. We transform G to an instance of Profit $\mathcal{P}$ as follows. $\mathcal{P}$ is defined on a directed bipartite graph

$G'(V_1, V_2, E')$, where $V_1 = V$ and $|V_2| = |E|$. For each edge $e = (u, v) \in E$, we draw a directed edge from vertex $e \in V_2$ to vertices $u \in V_1$ and $v \in V_1$. Thus in graph G', vertices in V_2 have no in-neighbors while vertices in V_1 share exactly one neighbor if and only if there exists an edge between the corresponding vertices in G.

We now claim the following: There exists a Hamiltonian Path in G if and only if there exists a profit solution in $\mathcal{P}$ of value exactly $n - 1$, where $|V| = n$. For the only if part, suppose there exists such a profit solution. Let Π_{opt} be the vertex relabeling of V_1 leading to this solution (vertices in V_2 need not be relabeled as they do not contribute to the profit). There are exactly n vertices in Π_{opt} and each vertex has a a profit of 1 with it's adjacent vertex, since all profits are at most 1. Then there must exist an edge in E corresponding to two consecutive vertices u_i, u_{i+1} in Π_{opt} since they have a profit exactly 1. Thus the labeling Π_{opt} corresponds to a Hamiltonian Path in G. For the if part, if there exists a Hamiltonian Path in G, then clearly the order of vertex traversal in this path corresponds to a labeling in $\mathcal{P}$ of net profit $n - 1$. ■

Since maximizing *Profit* over a graph is NP-hard, we will use a greedy heuristic to solve for $\mathcal{P}(G)$. Before we describe our solution, we expand our model to incorporate the fact that a cache can contain neighbors of not one, but multiple nodes depending on its size. After a total of i nodes have been processed, the *Profit* of j^{th} node should not be computed only with node i. Rather, it should be computed by matching $N_{in}(j)$ dynamically with the current cache contents, thus generating cache awareness in our metric. Let $Z(i)$ denote the contents of cache after i nodes have been processed. Then, *Profit* of j^{th} node is given by:

$$Profit(Z(i), j) = N_{in}(j) \cap Z(i) \tag{3}$$

If neighbors of m nodes are present in the cache, $Z(i)$ is determined by the union of all those neighbors. Note that m is not a constant and depends on the cache size and neighbor sets of previously processed nodes. This introduces additional challenges in the heuristic design, since we have to maintain the *Profit* relationship of graph vertices with a dynamic window of m nodes. Simultaneously, we also need to keep track of the cache contents $Z(i)$ and the set of m nodes whose neighbors reside in it. In further sections, we discuss the details of our heuristic and the data structures that we use to tackle these challenges.

IV. Heuristics

A. *Profit enhancing Heuristic (pH)*

pH is a greedy heuristic that solves for $\mathcal{P}(G)$ in an iterative manner as shown in Algorithm 2. It makes use of the fact that nodes are processed in the order of their indices and maintains a dynamic model of cache contents based on previously processed nodes. A new vertex labeling Π is therefore computed in a manner that matches the contents in our cache model with the neighbors of nodes to be processed. *pH* runs for $|V|$ iterations, each of which places one node consecutively in the new order Π. We denote the set of placed nodes as V_p and unplaced nodes as V_{up}. In the i^{th} iteration, *pH* picks an unplaced node with maximum *Profit* and assigns it an index value of i in Π. It then recomputes the cache contents $Z(i+1)$ for next iteration and evicts old nodes if $|Z(i + 1)|$ exceeds the cache capacity denoted by L.

Algorithm 2 *pH*

```
1:  while $i < |V|$ do
2:      $P_{max} = 0; v_{max} = 0;$
3:      for all $v \in V_{up}$ do
4:          if $Profit(Z(i), v) > P_{max}$ then
5:              $P_{max} = Profit(Z(i), v);$
6:              $v_{max} = v;$
7:      $\Pi[i] = v_{max};$
8:      $V_{up} = V_{up} - v_{max};$
9:      load : $Z(i + 1) = Z(i) \cup N_{in}(v_{max})$
10:     while $(|Z(i + 1)| > L)$ do
11:         evict : $Z(i + 1) = Z(i + 1) - N_{in}(i + 1 - m)$
12:         Update $m$
13:     Update $Profit(v', Z(i + 1)) \ \forall \ v' \in V_{up}$
14:     $i = i + 1$
```

pH uses a Least Recently Used (LRU) strategy for data eviction in the cache model similar to most real-world caches. It timestamps the usage of cached data and maintains a dynamic window of m recently processed nodes whose neighbors are in cache.

Initially, the cache is empty and neighbors of nodes being placed in Π will be simply augmented in the cache. Once it gets full, m is dynamically updated by evicting LRU neighbors of old nodes. The value of m is computed such that cache is maximally filled and neighbors of any other node cannot be accommodated in it. For a cache with capacity of L nodes, the value of m at the end of i^{th} iteration is given by:

$$m = \arg\max_k \ (|N_{in}(i) \cup N_{in}(i-1) \cup ... \cup N_{in}(i+1-k)| \leq L)$$

Once all nodes have been placed, we recompute the vertex array *VA[]* by accumulating the in degree of vertices in the order determined by Π. Edges in *EA[]* are relabeled and rearranged to assign them to their respective destination by placing them at the new offsets *VA[]*. The final CSR thus obtained, represents the reordered graph with optimized *Profit(G)* value.

Although *pH* enhances temporal locality, it has a drawback resulting from the fact that it considers individual node attributes as the quantum of data storage in cache. In reality, data is stored and evicted from cache in terms of cache lines that contain attributes of multiple graph vertices. To overcome this drawback, we develop a novel *Block Reordering* algorithm that is discussed in next subsection.

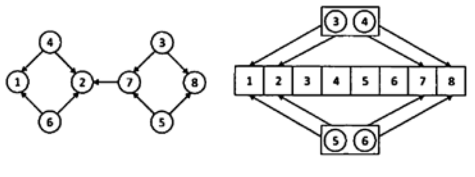
B. *Block Reordering*

When an array is laid out in memory, its elements are divided in different cache lines, each of which encompasses a fixed range of consecutive memory locations. We begin with a simple but critical observation: if any two locations belonging to different cache lines are accessed, then data has to be communicated twice from main memory to cache, ***irrespective of the relative closeness of these locations***. This observation leads us to the key idea behind *Block Reordering:* once the graph nodes are efficiently packed in cache lines, the order in which these cache lines are stored in main memory can be altered without degrading their utilization. *Block Reordering* exploits this fact to increase both spatial and temporal locality by permuting the graph nodes in two steps:

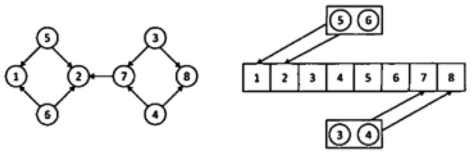
- The first step improves the spatial locality and cache line utilization for graph processing by packing nodes that are frequently accessed together, closer in the memory. We use GOrder [4] to perform this operation.
- In the second step, sets of consecutively indexed nodes in the reordered graph are grouped into blocks of equal size and merged together to create a *Hypernode*. Each *Hypernode* is connected with the same edges as its constituent nodes. Thereafter, we use *pH* to reorder *Hypernodes* and then expand them to generate the same structure as the original graph. Since, the contents of a *Hypernode* still have contiguous indices, spatial locality obtained after first step is preserved. The final graph order thus obtained, possesses high spatial and temporal locality and effectively utilizes both the features of the cache.

Fig. 3 illustrates the effect of *Block Reordering* on the same graph as fig. 2. Assume that one cache line can hold two nodes and capacity of cache is equal to one cache line. The 4 source nodes will therefore, be contained in 2 different cache lines. In the *pH* reordered graph shown in Fig. 3a, even though memory accesses are temporally localized, cache contention is high because only half of the cache line contents are used. Processing v_2 after v_1 will still incur 1 cache miss and so will processing v_8. In contrast, for the *Block Reordered* graph given in 3b, processing v_1 completely utilizes one cache line which is again reused by v_2. Thus, processing v_2 and v_8 leads to no cache misses.

Note that block size is a crucial input to *Block Reordering* algorithm. The reason why block size should be larger than the cache line size is that every CPU has its own mechanism to determine the alignment of data in main memory. The way we would partition nodes into cache lines for preprocessing, could be different from how it happens when the graph is loaded in DRAM. This could lead to one *Hypernode* being segregated into multiple cache lines and hence, loss of spatial locality. Hence, block size should be kept large enough for it to encompass multiple cache lines. This ensures that the contents of all cache lines that do not lie on the boundary of a block, are preserved. *Block Reordering* effectively trades off the granularity at which *pH* controls the permutation to preserve cache line constituents. If block size is too large, $P_{net}(G)$ after *pH* reordering will be low, leading to poor temporal locality. Therefore, block size should be carefully selected.



(a) *pH* Reordering



(b) Block Reordering

Fig. 3: Block Reordered and *pH* reordered graphs (left) and corresponding data layout in memory (right) with 2 nodes per cache line. Block Reordering increases both spatial and temporal locality.

V. Implementation

In this section, we discuss the implementation details of *pH* heuristic given in Algorithm 2. We first discuss naive implementations and evaluate the time complexity of reordering a graph. We observe that the cost of *pH* using existing data structures is prohibitively large and propose a new *Profit Bins* data structure to reduce the preprocessing time.

The most critical element in *pH* reordering is the *Profit* value. Assume that an array *P[]* stores the *Profit* of all nodes during any iteration. Recall from equation 3 that *Profit* of a node u is dependent on $N_{in}(u)$ and cache contents $Z(i)$ in that iteration. The former set remains constant but the latter changes as nodes are loaded and evicted from the cache. Hence, updates to *P[]* should be associated with updates to cache contents. In other words, *P[]* should be recomputed when nodes are loaded in or evicted from the cache.

Since *pH* is a cache aware algorithm, we need data structures to represent the cache contents $Z(i)$. Algorithm 2 performs two main operations on the cache: *load* and *evict*, which should be supported by these data structures. We fulfill these requirements using 3 components that together achieve an LRU cache model required for *pH* :

1) Cache Counter *(CC)* - an individual counter for every node. A zero *CC* value means that the node is not present in the cache. A non-zero counter value not only implies the presence of that node in cache but also represents the number of times that node was accessed while it was in cache. An array *CC[]* is used to store counter values for all the nodes in the graph.
2) *LRUptr* - a pointer to the oldest node whose neighbors are still in the cache.
3) Cache capacity L - the maximum number of nodes that can stay in cache at any time.

The following functions control the flow of data in our cache model while performing corresponding updates to the *Profit* values in *P[]*:

- FETCH(*CC[], n*) : Algorithm 3 represents how neighbors of a node n are added to the cache contents. For every in-neighbor u of node n, we check if u is already present in cache. If not, u is *loaded* into the cache and $P[v]\ \forall\ v \in N_{op}(u)\}$ is incremented. If u is already in cache, we only increment *CC[u]* to update the timestamp used to determine LRU data in cache.

Algorithm 3 Fill Cache

```
1: function FETCH($CC[], n$)
2:     for all $u \in N_{in}(n)$ do
3:         if ($CC[u] == 0$) then
4:             for all $v \in N_{op}(u)$ do
5:                 increase($P[v]$, 1);
6:         $CC[u] = CC[u] + 1$;
```

- REMOVE(*CC[], n*) : Algorithm 4 describes how neighbors of a node n are removed from our cache model. For every in-neighbor u of node n, we decrement the counter *CC[u]*. If *CC[u]* is still non-zero, it implies that u is not the least recently used node and hence, not evicted. If *CC[u]* becomes 0, u is *evicted* from cache and $P[v]\forall v \in N_{op}(u)$ is decremented by 1.

Algorithm 4 Empty Cache

```
1: function REMOVE($CC[], n$)
2:     for all $u \in N_{in}(n)$ do
3:         $CC[u] = CC[u] - 1$;
4:         if ($CC[u] == 0$) then
5:             for all $v \in N_{op}(u)$ do
6:                 decrease($P[v]$, 1);
```

Lemma 2. *The complexity of pH Reordering is*

$$O\Big(f(|V|)\cdot|V| \ + \ g(|V|)\cdot\sum_{u\in V} d^2_{op}(u)\Big)$$

where $f(x)$ is the complexity of removing the maximum amongst x numbers and $g(x)$ is the complexity of updating a value in a set of x numbers.

Proof. Consider the pseudocode for *pH* given in Algorithm 2. In every iteration, the algorithm pops the maximum *Profit* node and there are $|V|$ such iterations. Therefore, total time taken by the maximum popping operation is $O(f(|V|)\cdot|V|)$.

The second term arises from the *Profit* updates performed in *load()* and *evict()* operations. Consider the pseudocode given in Algorithm 3 and 4. When a node u is loaded in or evicted from the cache (i.e. *CC[u]=0*), *Profit* of all of its out-neighbors is incremented or decremented by 1. Therefore, time complexity of single transfer to or from cache is $O(g(|V|)\cdot d_{op}(u))$. In the worst case, number of loads and evictions of u are $\cdot d_{op}(u)$. If every node in the graph encounters the worst case, number of *Profit* updates performed are $O(d^2_{op}(u))$ and the cost of processing the entire graph is $O(g(|V|)\cdot\sum_{u\in V} d^2_{op}(u))$. ■

Naive Implementations: The most straightforward way to implement Algorithm 2 is to use a single array *P[]* to store the *Profit* value. In this case, $f(x) = O(x)$ and $g(x) = O(1)$. Plugging these values in Lemma 2, we obtain the reordering time to be $O(|V|^2 + \sum_{u\in V} d^2_{op}(u))$. For large graphs, this is prohibitively expensive due to the $|V|^2$ term.

To reduce the complexity of popping the maximum value, we can implement *P[]* as a max-heap giving $f(x) = O(\log(x))$ and $g(x) = O(\log(x))$. The reordering time for this implementation would be $O(\log(|V|)\cdot(|V|+\cdot\sum_{u\in V} d^2_{op}(u)))$. Even for a graph with 1 million vertices, $\log(|V|) = 20$ and the reordering becomes very costly. Therefore, none of these data structures can be used to implement *pH* reordering with viable computation time for large graphs.

A. *Profit Bins*

To make *pH* reordering viable, we develop a new *Profit Bins* data structure which comprises of an additional header array called *Bins[]*. Just like *VA[]* in CSR graph format, *Bins[i]* is offset to the location of first node with *Profit* value i in the *P[]* array. The *P[]* array is always kept sorted so that nodes with same *Profit* are stored consecutively in the same *Bin* and maximum *Profit* node is always located at the end of the array. This data structure is shown in fig. 4. We use a pointer *maxPtr* to denote the end of the array which shifts when the maximum node is popped out. Nodes with index less than *maxPtr* in *P[]* array represent the set of unplaced nodes V_{up}.

Note that in Algorithm 3 & 4, *P[v]* is incremented or decremented by 1 only. We makes use of this fact to support constant time updates to *P[]* array. The pseudocode for *increase()* and *decrease()* operations is given in Algorithm 5. Note that to perform the swap operation in constant time, we also maintain a mapping of vertex v to its position in *P[]* array.

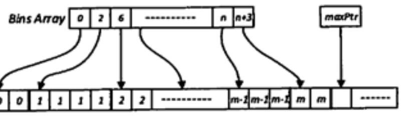


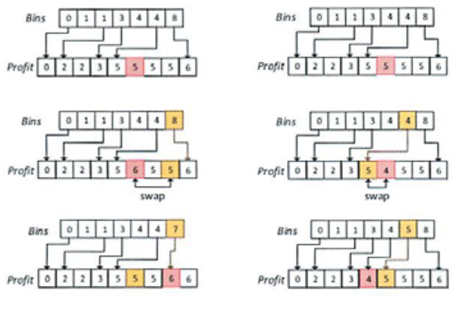
Fig. 4: Optimized Data Structure for *pH* implementation.

We first check if $v \in V_{up}$ so that nodes that are already placed are not processed again. If *P[v]* is to be incremented by 1, we shift it to the beginning of the next bin; if *P[v]* is to be decremented by 1, we shift it to the end of preceding bin. Offsets of the bins are adjusted accordingly before the function returns. Fig. 5 illustrates the update operations on *Profit Bins* data structure.

With the *Profit Bins* data structure, both updates and maximum node popping are executed in constant time i.e. $f(x) = O(1)$ and $g(x) = O(1)$ drastically reducing the preprocessing time. Moreover, the complexity of reordering given

Algorithm 5 *Profit* Updates

```
function INCREASE($v$, $P[]$, $Bins[]$)
    if $v < maxPtr$ then
        $P[v] = P[v] + 1$;
        currBin = $P[v]$;
        newPos = $Bins$[currBin] $- 1$;
        swap($v$, newPos);
        $Bins$[currBin] = $Bins$[currBin] $- 1$;
function DECREASE($v$, $P[]$, $Bins[]$)
    if $v < maxPtr$ then
        currBin = $P[v]$;
        $P[v] = P[v] - 1$;
        newPos = $Bins$[currBin];
        swap($v$, newPos);
        $Bins$[currBin] = $Bins$[currBin] $+ 1$;
```



(a) Increase *P[5]* (b) Decrease *P[5]*

Fig. 5: Update operations in *Profit Bins* data structure.

in Lemma 2 is a pessimistic upper bound. In practice, a node u will not be loaded from cache by each of its out-neighbors. If $R(d_{op}(u))$ denotes the ratio of number of times u is loaded to its out degree, complexity of reordering is given by:

$$O\Big(|V| + \sum_{u \in V} R(d_{op}(u)) \cdot d_{op}^2(u)\Big)$$

Fig.6 shows the variation in $R(d_{op}(u))$ with the in-degree of node u obtained by pre-processing Google+ dataset. We observe that $R(d_{op}(u)) < 0.05$ for most high degree nodes.

VI. Evaluation

A. Target Applications

We implement and evaluate our reordering over 4 diverse algorithms that are extensively used in a variety of graph processing applications. A description of the target algorithms is given below:

- **SpMV Kernels, Pagerank:** Sparse-matrix vector multiply is a fundamental kernel used by a large class of algorithms [21]. In this paper, we use Pagerank [22] as a representative algorithm to evaluate performance of

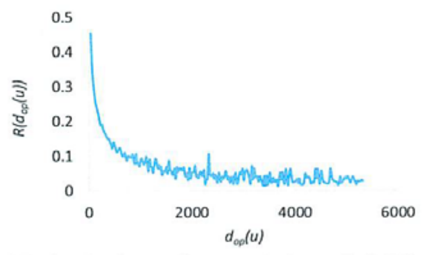


Fig. 6: Ratio of cache transfers vs node degree. Probability of eviction reduces as degree increases.

  SpMV kernels. It iteratively compute $x^{t+1} = Ax^t + B$ where A is the adjacency matrix of graph and x is pagerank vector.
- **Label Propagation:** Label propagation is widely used to study spread of influence in graphs and node classification [23]. In our example, each node is initialized with a distinct label and iteratively selects the minimum label among its neighbors. This particular approach is used to detect connected components in a graph [24].
- **Community Detection:** Many algorithms that find community structures in graphs [25], [26], [27] can be classified as stationary. For the purpose of illustration, we implement the algorithm given in [27] which iteratively associates a node with the community to which majority of its neighbors belong.
- **Single Source Shortest Path (SSSP):** SSSP is a key kernel in the Graph 500 benchmark list [1]. We use Bellman-Ford algorithm to implement SSSP kernel.

B. Datasets

We use 8 real world datasets with size ranging from 1 million to 95 million nodes and 16.5 million to 1.94 billion edges. The graph data is stored in CSR format [10]. Table I summarizes size and sparsity characteristics of these datasets.

TABLE I: Real world Graph datasets

Dataset	# Vertices	# Edges	Average degree
Pokec [28]	1.6 M	30.6 M	19.1
CitPatent [28]	3.7 M	16.5 M	4.5
LiveJournal [29]	4.8 M	69 M	14.4
WikiLink [29]	12.1 M	378 M	31.2
Google+ [30]	28.9 M	463 M	16
Pld [31]	43 M	623 M	14.5
FbKonect [32]	59 M	186 M	3.2
Sd1 [31]	94.9 M	1937 M	20.4

Pokec, LiveJournal, Google+ and Fbkonect are social networks; CitPatent is a patent citation network; WikiLink, Pld and Sd1 are hyperlink graphs. Google+ is the largest social network graph and Sd1 is the largest web graph among these datasets. CitPatent and FbKonect are extremely sparse in nature with average degree < 5.

TABLE II: Execution time of Pagerank (in seconds)

Dataset	Original	In degree FC	Out degree FC	GOrder	*pH*	Block Reordering
Pokec	40.1	42	44.3	33.9	29.5	**28.7**
Citpatent	40	39	27.9	31.5	28.7	**24.1**
LiveJournal	85.9	102.6	103.9	73.1	69.4	**65.4**
Wikilink	315	376	410	270	264	**247**
Google+	866	950	976	612	563	**523**
Pld	1268	1164	1269	823	749	**716**
FbKonect	366	217	209	211	169	184
Sd1	3988	3635	4604	1988	2267	**1728**

C. Experimental Setup

We conducted experiments on a linux server equipped with AMD Opteron 6278 CPU@2.4GHz running Ubuntu 14.04 operating system. The per core L1 and L2 cache sizes for this CPU are 76KB and 2MB respectively, and the shared L3 cache size is 6MB. All code is written in C++ and compiled using G++ 4.7.1 with highest optimization -O3 flag. The L1 and L3 cache statistics are collected using the *perf* tool.

We set the cache capacity $L = 100k$ and block size to 20 for *pH* and *Block Reordering* heuristics. These parameter values are empirically determined on our server by analyzing the pre-processing time and cache performance of reordered graphs. A thorough design space exploration of these parameters is of independent interest but beyond the scope of this paper due to space considerations.

Baselines: We compare our reordering against the state-of-the-art GOrder algorithm [4], and the Frequency based Clustering (FC) [9] that was recently proposed. In [4], it is shown that the performance of GOrder is superior to reordering algorithms previously developed like RCM [17], CHDFS [18] and Minimum Linear and Logarithmic Arrangements [33] and hence, we do not compare against these algorithms. Cache line size parameter for GOrder is set to the optimum value of 5 as specified in [4]. Frequency based clustering is performed using both in degree and out degree of graph nodes.

D. Results

Execution Time Speedup: We use execution time as the main metric to analyze the performance of different reorderings. The execution time is measured by running Pagerank, Label Propagation and Community Detection for 100 iterations and SSSP for 10 different starting nodes. We further repeat each of these experiments 5 times and report the average running time. Table II compares the runtime of Pagerank on all 8 datasets with different node orderings. We see that *Block Reordering* consistently outperforms *pH* and the baselines for all the datasets except FbKonect. We also observe that performance of FC is not consistent and its speedup for various datasets is low with the execution time being even worse than that of the original order.

Fig. 7 gives a comparison of the percentage speedup obtained per iteration by *pH* and *Block Reordering* against the baseline for all target applications. Percentage speedup of a reordered graph is computed as $100 * \left(\frac{T_{orig}}{T_{re}} - 1\right)$, where T_{orig} and T_{re} are the execution time per iteration of an application on original and reordered graphs, respectively. We see that apart from FbKonect, FC fails to obtain significant improvement on other datasets. GOrder achieves high performance specially on large graph datasets where it accelerates the processing by 50% to 100%. However, the speedup obtained by *pH* alone is comparable to GOrder in most cases. Further, by performing joint optimization, *Block Reordering* outperforms both *pH* and GOrder. We also observe that the performance enhancement of *Block Reordering* and *pH* is particularly high for very large graphs. This is because for large graphs, very small fraction of nodes can fit in the cache and therefore, original graph order incurs large number of cache misses.

Cache Miss Ratio: Table III and IV show the L1 and L3 cache statistics during execution of Pagerank on Google+ and Sd1 dataset, respectively. The number of memory references and L3 misses are represented in billions (B). Total memory references are the same as the L1 references reported by *perf* since every access first goes through the L1 cache. It is evident that while GOrder reduces the L1 cache misses, *pH* efficiently decreases L3 cache miss ratio. *Block Reordering* combines both the ideas to effectively minimize the total number of L3 cache misses resulting in reduced communication with main memory. It incurs approximately 20% to 25% less L3 cache misses than GOrder and *pH*. Table V and VI give the overall cache miss ratio ($\frac{\text{LLC misses}}{\text{Total references}}$) on Google+ and Sd1 dataset, respectively, during execution of all the target algorithms listed in Section VI-A. It is evident that for all the algorithms, *Block Reordering* achieves minimum cache miss ratio amongst all the graph orders.

TABLE III: Cache performance for Pagerank on Google+

Graph Order	# refs	L1-mr	L3-mr	# L3 misses
Original	126.5 B	29.4%	68%	26.9 B
In degree FC	131.9 B	33.6%	70.4%	32.8 B
Out Degree FC	132.6 B	33.9%	72.9%	34.3 B
GOrder	122.5 B	16%	54.6%	12.7 B
pH	123.1 B	27%	33.6%	12.1 B
Block Reordering	123.4 B	17.4%	41.6%	10.2 B

Pre-processing Time: Table VII shows the time taken to compute all the reorderings compared above. In-degree and Out-degree FC have the smallest pre-processing cost because they are essentially performing a sort operation on

https://github.com/datourat/Gorder

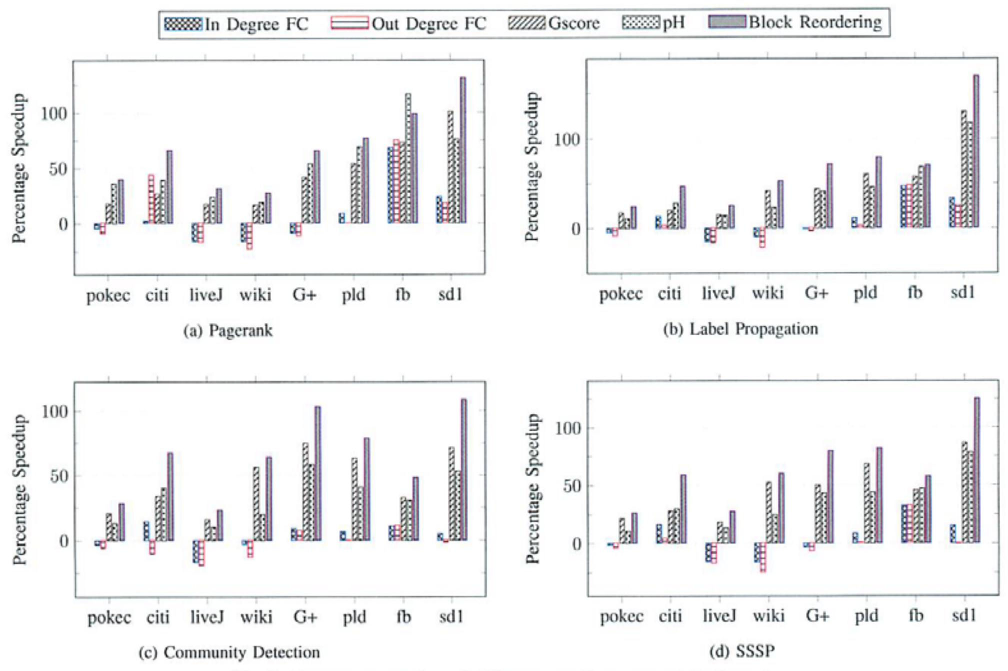


Fig. 7: Percentage speedup of different orderings over original graph.

TABLE IV: Cache performance for Pagerank on Sd1

Graph Order	# refs	L1-mr	L3-mr	# L3 misses
Original	530.6 B	34.2%	68.9%	132.4 B
In degree FC	534 B	25.3%	67.5%	99.2 B
Out Degree FC	531 B	32.3%	56.3%	102.7 B
GOrder	486.4 B	15%	45.1%	42.3 B
pH	487.8 B	28.9%	29.2%	44.2 B
Block Reordering	491 B	16.9%	36.1%	34.9 B

TABLE V: Overall cache miss ratio for Google+ dataset

Graph Order	Pagerank	Label Propagation	Community Detection	SSSP
Original	21.3%	18.6%	4.9%	14%
In degree FC	24.9%	21.6%	4.9%	16.3%
Out degree FC	25.9%	22.9%	5.5%	17.1%
GOrder	10.3%	8.6%	2.2%	6.5%
pH	9.8%	7.9%	2%	6.1%
Block Reordering	**8.2%**	**6.3%**	**1.7%**	**4.9%**

TABLE VI: Overall cache miss ratio for Sd1 dataset

Graph Order	Pagerank	Label Propagation	Community Detection	SSSP
Original	25%	23.2%	5.7%	17.5%
In degree FC	18.6%	16.7%	4.1%	12.5%
Out degree FC	19.4%	18.6%	4.6%	14.1%
GOrder	8.7%	6.9%	1.9%	5.1%
pH	9.1%	7.9%	2.2%	5.8%
Block Reordering	**7.1%**	**5.7%**	**1.6%**	**4.4%**

graph nodes. However, as observed earlier, FC is unable to reduce the cache misses consistently for all the graph datasets. Moreover, node indexing is computed offline only once. Therefore, even though the other reordering techniques have a high pre-processing cost, their savings are huge as the target applications are run multiple times on the reordered graph amortizing the reordering cost.

While the speedup obtained by *pH* is comparable to the state of the art GOrder algorithm, its reordering time is less than half that of the latter. Note that the time reported for *Block Reordering* does not include the cost of computing GOrder. From Table VII, we see that its overhead is only 10% to 20% above GOrder for all the graphs.

VII. Conclusion

Due to the irregularity in data access patterns of graph analytic algorithms, caches in CPU systems fail to reduce the main memory accesses effectively. In this work, we presented a novel *Block Reordering* algorithm that enhances both spatial and temporal locality in access patterns of graph algorithms to comprehensively address the issue of cache performance.

TABLE VII: Reordering time (in seconds)

Dataset	In degree FC	Out degree FC	GOrder	*pH*	Block Reordering
Pokec	0.7	0.75	31.3	14	5.9
CitPatent	0.98	1	9.9	4.9	1.44
LiveJournal	1.85	1.84	94	28.9	11
WikiLink	6.26	6.41	736	239	57
Google+	14.8	17.1	1597	650	197
Pld	29.3	24.8	3027	1705	443
FbKonect	13.7	13.6	257	55.9	35.7
Sd1	63.2	52.9	11651	3791	2754

Experimental results show that our reordering can achieve upto 2.3× performance compared to original graph order while consistently outperforming the state of the art GOrder algorithm by 20% to 25% reduction in cache misses. This highlights the necessity of combined spatio-temporal locality enhancement to achieve high cache performance for graph processing. We believe that sophisticated algorithms can be developed that perform such joint optimization in a single reordering step. This can drastically reduce the pre-processing cost and perhaps, result in even better performance.

This paper also delineates the development of a parametrized cache-aware reordering algorithm *pH*. It takes into account the capacity and the eviction policy of cache. Although *pH* is designed for LRU caches, the underlying methodology can be adapted for other memories such as user controlled on-chip memory on GPU and FPGA. Being parametrized, the performance of *Block Reordering* and *pH* is sensitive to the choice of cache capacity and block size values. In future work, we intend to determine the optimal range of values of these parameters by rigorously exploring their effect on performance.

Acknowledgements: We would like to thank the anonymous reivewers whose insightful feedback helped in improving this paper. We are also grateful to Dr. Oded Green and Dr. David Bader from Georgia Institute of Technology for sharing their expertise in the technical discussions we had during the course of this research.

REFERENCES

[1] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," 2010.

[2] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, 2007.

[3] S. Beamer, K. Asanovic, and D. Patterson, "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2015.

[4] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup Graph Processing by Graph Ordering," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016.

[5] L. Auroux, M. Burelle, and R. Erra, "Reordering Very Large Graphs for Fun & Profit," in *International Symposium on Web AlGorithms*, 2015.

[6] P. Boldi and S. Vigna, "The Webgraph Framework I: Compression Techniques," in *Proceedings of the International Conference on World Wide Web*, 2004.

[7] P. Z. Chinn, J. Chvtalov, A. K. Dewdney, and N. E. Gibbs, "The bandwidth problem for graphs and matrices-a survey," *Journal of Graph Theory*, 1982.

[8] L. Harper, "Optimal numberings and isoperimetric problems on graphs," *Journal of Combinatorial Theory*, 1966.

[9] Y. Zhang, V. Kiriansky, C. Mendis, M. Zaharia, and S. P. Amarasinghe, "Optimizing Cache Performance for Graph Analytics," *CoRR*, vol. abs/1608.01362, 2016.

[10] "Boost – c++ libraries," 2005, available at http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/compressed_sparse_row.html.

[11] Y. Guo, S. Hong, H. Chafi, A. Iosup, and D. Epema, "Modeling, analysis, and experimental comparison of streaming graph-partitioning policies," *Journal of Parallel and Distributed Computing*, 2016.

[12] X. Wang, Y. Zhu, and Y. Chen, "Quantitative Analysis of Graph Algorithms: Models and Optimization Methods," in *IEEE BigDataSecurity, HPSC and IDS*, 2016.

[13] M. Penner and V. K. Prasanna, "Cache-Friendly Implementations of Transitive Closure," *J. Exp. Algorithmics*, 2006.

[14] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing memory performance for fpga implementation of pagerank," in *ReConFigurable Computing and FPGAs, International Conference on*. IEEE, 2015.

[15] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing Sparse Matrix-Vector Multiplication for Large-Scale Data Analytics," in *Proceedings of the International Conference on Supercomputing*, 2016.

[16] M. Kumar, M. Serrano, J. Moreira, P. Pattnaik, W. P. Horn, J. Jann, and G. Tanase, "Efficient implementation of scatter-gather operations for large scale graph analytics," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2016.

[17] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali, "Parallelization of reordering algorithms for bandwidth and wavefront reduction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[18] J. Banerjee, W. Kim, S. J. Kim, and J. F. Garza, "Clustering a DAG for CAD databases," *IEEE Transactions on Software Engineering*, 1988.

[19] A. Abou-Rjeili and G. Karypis, "Multilevel Algorithms for Partitioning Power-law Graphs," in *Proceedings of the International Conference on Parallel and Distributed Processing*, 2006.

[20] A. Mukkara, N. Beckmann, and D. Sanchez, "Cache-Guided Scheduling: Exploiting Caches to Maximize Locality in Graph Processing," in *1st International Workshop on Architecture for Graph Processing*, 2017.

[21] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High Performance Graph Analytics Made Productive," *Proc. VLDB Endowment*, Jul. 2015.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," 1998.

[23] X. Zhu and Z. Ghahramani, "Learning from Labeled and Unlabeled Data with Label Propagation," Tech. Rep., 2002.

[24] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2012.

[25] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, 2008.

[26] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010.

[27] X. Liu and T. Murata, "Advanced modularity-specialized label propagation algorithm for detecting communities in networks," *Physica A: Statistical Mechanics and its Applications*, 2010.

[28] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, Jun. 2014.

[29] J. Kunegis, "KONECT: The Koblenz Network Collection," in *Proceedings of the International Conference on World Wide Web*, 2013.

[30] N. Z. Gong, W. Xu, L. Huang, P. Mittal, E. Stefanov, V. Sekar, and D. Song, "Evolution of Social-attribute Networks: Measurements, Modeling, and Implications Using Google+," in *Proceedings of the Internet Measurement Conference*, 2012.

[31] O. Lehmberg, R. Meusel, and C. Bizer, "Graph Structure in the Web: Aggregated by Pay-level Domain," in *Proceedings of the ACM Conference on Web Science*, 2014.

[32] R. A. Rossi and N. K. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2015.

[33] I. Safro and B. Temkin, "Multiscale approach for the network compression-friendly ordering," *CoRR*, vol. abs/1004.5186, 2011.

A-2. Shreyas G. Singapura, Rajgopal Kannan, Viktor K. Prasanna, On-chip Memory Efficient Data Layout for 2D FFT on 3D Memory Integrated FPGA, IEEE 20th Annual High Performance Extreme Computing Conference (HPEC), pp. 1-7, September 2016

On-chip Memory Efficient Data Layout for 2D FFT on 3D Memory Integrated FPGA

Shreyas G. Singapura, Rajgopal Kannan and Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California, Los Angeles, CA 90089, USA
{singapur, rajgopak, prasanna}@usc.edu

***Abstract*—3D memories are becoming viable solutions for the memory wall problem and meeting the bandwidth requirements of memory intensive applications. The high bandwidth provided by 3D memories does not translate to a proportional increase in performance for all applications. For an application such as 2D FFT with strided access patterns, the data layout of the memory has a significant impact on the total execution time of the implementation. In this paper, we present a data layout for 2D FFT on 3D memory integrated FPGA that is both on-chip memory efficient as well as throughput-optimal. Our data layout ensures that consecutive accesses to 3D memory are sufficiently interleaved among layers and vaults to absorb latency due to activation overheads for both sequential (Row FFT) and strided (Column FFT) accesses. The current state-of-the-art implementation on 3D memory requires $O(\sqrt{c}N)$ on-chip memory to reduce the strided accesses and achieve maximum bandwidth for an $N \times N$ FFT problem size and c columns in a 3D memory bank row. Our proposed data layout optimizes the throughput of both the Row FFT and Column FFT phases of 2D FFT with $O(N)$ on-chip memory for the same problem size and memory parameters without decreasing the memory bandwidth thereby achieving a $\sqrt{c}\times$ reduction in on-chip memory. On architectures with limited on-chip memory, our data layout achieves $2\times$ to $4\times$ improvement in execution time compared with the state-of-art 2D FFT implementation on 3D memory.**

I. Introduction

3D memory is a potential solution to the memory wall problem of low bandwidth with a promising bandwidth in the range of 300-400 GB/s [1], [2]. Although 3D memories provide higher bandwidth than existing 2D memory technologies such as DDR3 [3], the actual bandwidth available depends on the access pattern of the application and thus may be much lower than the peak bandwidth. In particular, the traditional row activation overhead can become a bottleneck due to random accesses to multiple rows. To maximize the available bandwidth from the 3D memory, optimizations specific to the access pattern of the application need to be developed.

Fast Fourier Transform (FFT) is an important kernel used in signal processing applications [4], [5]. Specifically, 2D FFT is widely used in image processing applications and requires high throughput for large image sizes [6], [7]. 2D FFT is implemented as a combination of Row and Column FFT phases [8]. The Row FFT phase consists of sequential accesses to the memory which result in high bandwidth and low latency. On the other hand, the Column FFT phase has strided accesses which translate to accesses to different rows of a bank. These non-sequential accesses to the memory result in low bandwidth and high latency due to the activation overhead of accessing multiple rows.

There have been works [9], [10] targeting optimizations to the data layouts for 2D FFT implementation on 3D memory. The previous works focus on achieving high bandwidth by storing multiple rows/columns of input data in on-chip memory and reducing number of strided accesses to the memory. Although high bandwidth is achieved by the proposed data layout, large number of complete rows/columns need to be stored in on-chip memory to achieve this high bandwidth. As the problem size increases, the amount of on-chip memory has to be increased proportionally to store the entire rows/columns and maintain high bandwidth. With limited on-chip memory, the data layouts of existing works is not able to extract maximum bandwidth from the 3D memory which results in higher execution time.

We observe that the structure of 3D memory can be exploited to hide the latency overhead of accessing multiple rows resulting from strided accesses. By employing the inter-layer pipelining and parallel vault access features of 3D memory, we develop an optimized data layout that provides maximum bandwidth for both sequential and strided accesses and requires only the necessary elements to be stored in on-chip memory. Our throughput-optimal data layout for 2D FFT on 3D memory has the following key features: (1) Consecutive elements are mapped to different layers to exploit faster access across 3rd dimension than mapping elements to the same layer; (2) Vaults are accessed in parallel since the latency of multiple vault access is the same as that to a single vault and (3) Accesses to the same layer are separated by sufficient number of intermediate accesses to other layers to hide the access latency. The main contributions of this paper are:

- We present an optimized data layout for 2D FFT on 3D memory that achieves maximum bandwidth for both sequential and strided access patterns of 2D FFT.
- Our data layout achieves an on-chip memory requirement of $O(N)$ for a 2D FFT problem size of $N \times N$ without sacrificing the bandwidth and latency of 3D memory.
- Our data layout achieves $\sqrt{c}\times$ reduction in on-chip memory compared with state-of-the-art 2D FFT implementation given c columns in a row of a memory bank.
- On architectures with limited on-chip memory, our data

This material is based in part upon work supported by the Air Force Research Laboratory under Grant Number FA8750-15-1-0185.

layout achieves $2\times$ to $4\times$ improvement in execution time compared with state-of-the-art 2D FFT implementation for large problem sizes.

The rest of the paper is organized as follows. Section II covers the related work for 2D FFT on 3D memories. Section III describes the target architecture and its components. Section IV describes the Baseline data layout and introduces the proposed Optimized data layout. Section V presents the evaluation methodology and performance analysis. Section VI concludes the paper.

II. Background and Related Work

2D FFT can be implemented in two phases of Row FFT and Column FFT using the row-column algorithm by performing 1D FFTs on rows and columns of inputs [8]. In the Row FFT phase of the algorithm, for a problem size of $N \times N$ input matrix, 1D FFT is applied on each row of the input matrix in sequential order. The outputs of the Row FFT phase act as inputs to the Column FFT phase. In the Column FFT phase, 1D FFT is applied on each column of the $N \times N$ matrix and the outputs of Column FFT phase represent the final output of 2D FFT on the original $N \times N$ input matrix.

2D FFT on 3D memory has been the focus of many research works. In [9], memory optimized data layouts is developed for FFT on hardware accelerators such as ASIC and FPGA. Block data layout is implemented for DDR3 memory and later extended to 3D memory. A block is mapped to a row of a bank and multiple blocks are distributed among banks to increase the bandwidth of the memory. For a block of size $t \times t$ and a problem size $N \times N$, the on-chip memory requirement is of the order $O(tN)$. In [11], a Logic-in-Memory (LiM) IC is developed to perform 2D FFT on 3D memory. Application specific logic cores are used to implement 2D FFT and energy efficiency and bandwidth are targeted as the performance metrics. Although inter-layer pipelining is utilized, block data layout from [9] is used. In [10], processing kernel on FPGA is developed to implement dynamic data layouts to reduce the number of row activations. Multiple rows/columns (p) of input data are prefetched from the memory and permutation network is used while writing back the outputs to memory to reduce the number of row activations. The on-chip memory requirement is of the order $O(pN)$, for $1 < p < t$. None of these works focus on the on-chip memory and require substantial amount of on-chip memory to achieve high bandwidth for large problem sizes.

In this paper, we propose an Optimized data layout to implement 2D FFT on 3D memory which achieves a minimum on-chip memory requirement without sacrificing the bandwidth and latency of 3D memory. We exploit inter-layer pipelining and parallel vault access to hide the latency of accessing elements in the same layer and overhead of accessing multiple rows. By achieving maximum bandwidth for both Row and Column FFT phases, our data layout stores only the necessary elements in on-chip memory and minimizes the on-chip memory requirement.

III. Target Architecture

Our target architecture is a 3D memory integrated FPGA consisting of 3D memory and an FFT Processing Unit (PU) on FPGA. The components of the architecture are illustrated in Figure 1.

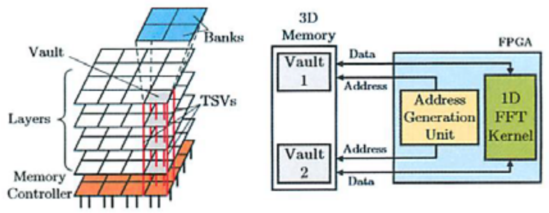


Fig. 1: (a) Architecture of a 3D Memory (b) FFT PU on FPGA

A. 3D Memory

3D memory is organized as a set of v vaults consisting of l layers and b banks per layer in a vault. Data in a vault is accessed using vertical interconnects (TSVs). A representative archtiecture of 3D memory consisting of 16 vaults with 4 layers and 4 banks per layer is illustrated in Figure 1(a). Vaults do not share TSVs with one another and hence can be accessed in parallel. Within a vault, data in different layers can be accessed at a faster rate than data in the same layer, a property known as *inter-layer pipelining* [12], [11], [13]. This is because the latency of activation overhead of rows in different layers can be overlapped due to fast TSVs. Within a layer, the structure of 3D memory is similar to the structure of DDR3 with data stored in rows and columns in each bank. Accessing data stored in different rows of the same bank incurs large latency due to row activation overhead whereas, bank interleaving can be used to reduce the latency by accessing data in different banks. Each data element stored in a 3D memory can be accessed by specifying the address in terms of vault, layer, bank, row and column. For each read/write request to the 3D memory, a specific row in a bank belonging to a layer in a vault is accessed and the bandwidth and latency of 3D memory depends on the access pattern of these requests.

In our previous work [10], [14], [15], we developed a parameterized model of the 3D memory to identify the parameters which have a significant impact on the bandwidth and latency of 3D memory. Our model characterizes the 3D memory in terms of timing parameters which take into account the architecture and different access patterns. For the sake of completeness, we describe once again the parameters of the 3D memory model:

- t_{vault}: time between accesses to *different vaults*
- t_{layer}: time between accesses to *different layers* in a vault
- t_{bank}: time between accesses to *different banks* in a layer in the same vault
- t_{row}: time between accesses to *different rows* in a bank
- t_{col}: time between accesses to *different columns* in a row

It should be noted that all the above parameters except t_{row} are defined assuming the rows being accessed are already active.

B. FFT Processing Unit (PU) on FPGA

The FFT processing unit consists of a 1D FFT kernel and an address generation unit. 1D FFT kernel processes inputs of size N to produce FFT outputs. The address generation unit maps the inputs and outputs of the 1D FFT kernel to the required addresses in the memory. Since the kernel can process streaming data, we use different vaults to read inputs and write the outputs. In Figure 1, in the Row FFT phase, Vault 1 acts as the input vault and Vault 2 acts as the output vault. In the Column FFT phase, their roles are reversed.

IV. Data Layouts

In this section, we describe the Baseline data layout and its limitations. Later, we present our proposed Optimized data layout along with the mapping technique. The parameters of the architecture used in our analysis and their definitions are described in Table I. We assume each access to a vault results in a column of data being available from the memory. For notation convenience, we assume there are $2v$ vaults in the memory; v vaults are used to read inputs and v vaults are used to write the outputs. This assumption does not affect our proposed data layout or the performance analysis.

Notation	Definition
$N \times N$	problem size
$2v$	# vaults in 3D memory
l	# layers in a vault
b	# banks per layer in a vault
r	# rows in a bank
c	# columns in a row of a bank

TABLE I: Parameters of 3D Memory

A. Baseline Data Layout

We use the block data layout proposed in [9] as the Baseline data layout. In this data layout, a block or a tile of size $t \times t$ is mapped to a row of a bank in the memory and multiple such blocks are mapped to different banks. The value of t ranges between $[1, \sqrt{c}]$. In order to perform a 1D FFT of a row of N elements, an entire row of blocks (equivalent to t rows of $N \times N$) are transferred from the 3D memory to on-chip memory. An FFT kernel is used to process the data stored in on-chip memory and the outputs are written back to memory. This process is repeated for all the rows in the input matrix to complete the Row FFT phase. In the subsequent Column FFT phase, entire column of blocks are transferred to the on-chip memory and processed to produce the final Column FFT outputs. Although the Baseline data layout can enable high throughput, we observe the following limitations of this data layout.

Limitation 1: Bandwidth of the 3D memory is proportional to the block size with $t^2 = c$ achieving maximum bandwidth.

For $t^2 = c$, blocks are accessed from different layers and the latency overhead of accesses to the same bank is overlapped with accesses to banks in other layers and the bandwidth is limited by t_{layer}. For $t^2 < c$, majority of the consecutive blocks are mapped to banks in the same layer and t_{bank} and t_{col} will limit the bandwidth of the 3D memory. The effect of small block sizes on performance is evident in [9], with $t = (4, 8)$ achieving $(33\%, 50\%)$ of the performance in comparison with that of $t = 32$.

Limitation 2: For an $N \times N$ problem size, $O(\sqrt{c}N)$ on-chip memory is required to achieve maximum bandwidth.

At any point of time, an entire row/column of blocks of data (tN elements) need to be stored in on-chip memory to process N elements of a row/column. Based on Limitation 1, maximum bandwidth is achieved for $t^2 = c$. Therefore, the on-chip memory required is $\sqrt{c}N$ elements of data. In [9], the authors use block size $t = 32$ to achieve maximum bandwidth. For problem sizes $N = [8192, 32768]$ complex single-precision (2×32 bits per word) inputs, this translates to a large on-chip memory requirement in the range of $16-67$ Mbits.

Therefore, the Baseline data layout requires large on-chip memory to achieve maximum bandwidth from 3D memory and on limited on-chip memory architectures, bandwidth of 3D memory reduces which translates to higher execution time.

B. Optimized Data Layout

Our data layout is defined by two mapping functions, corresponding to each phase of 2D FFT. Each function is a mapping of an $N \times N$ matrix to locations in 3D memory. A location (address) is defined by the quintuple $\{v(a_{ij}), l(a_{ij}), b(a_{ij}), c(a_{ij}), r(a_{ij})\}$ which maps matrix element a_{ij} to a vault, layer, bank, column and row in the 3D memory. The first mapping function (DL 1) describes the layout of FFT input matrix A in 3D memory before the start of the Row FFT phase. The second mapping function (DL 2) is used to write the elements of matrix $\widehat{A}$, the output of the Row phase, to 3D memory. The same layout is then used to read columns of $\widehat{A}$ during the Column FFT phase. The outputs of this phase are the final outputs and can follow either data layout above, depending on how the resultant matrix is to be used further.

In order to derive our mapping scheme for optimal on-chip storage and bandwidth maximizing 2D FFT data layout, we make the following basic assumption about the timing parameters of 3D memory: $t_{layer} \leq \{t_{bank}, t_{col}\} \leq t_{row}$. We also assume the number of layers is sufficient to make $l \cdot t_{layer} \geq \{t_{col}, t_{bank}\}$ (we also describe the performance results in Section V-B when this assumption is relaxed). These assumptions are based on our estimates of the timing and architecture parameters of 3D memory, as described in [1]. The 3D memory in [1] has a peak bandwidth of 8 GB/s per vault and an element of 64 bits can be accessed for each memory request, which translates to an access time of 1 ns for each element. Therefore, we assume $t_{layer} = 1\ ns$ which represents the least possible latency of memory accesses. Further, since the structure of a layer in a 3D memory is similar to DDR3 [3], we estimate the values of other timing parameters as $t_{bank} = 2\ ns$, $t_{col} = 4\ ns$ and $t_{row} = 40\ ns$ based on timing parameters described in [3].

The key characteristics of our mapping scheme based on the above assumptions are as follows: Since vaults can be accessed in parallel, it is trivial to distribute elements across vaults to maximize bandwidth. Our data layout further maps accesses within a vault to different layers to ensure the minimum possible latency of t_{layer} for each access. Now, considering accesses within a vault, our layout maximizes bandwidth by hiding the latency of consecutive accesses to the same row or different rows in a bank through a number of intermediate accesses to other layers, utilizing $p \cdot t_{layer} \geq t_{col}$ and $q \cdot t_{layer} \geq t_{row}$. For example, choosing $p \geq 4$ and $q \geq 40$ based on the parameters above, will hide the latency of t_{col} and t_{row} and incur a minimum latency of t_{layer}. Hiding the latency of accesses to different rows and columns is possible due to the large number of banks [1] and faster access across the 3rd dimension of 3D memory [11], [13]. We prove that our data layout minimizes the latency in Section V-B.

For notational simplicity and WLOG, in our description of the mapping schemes, we assume that parameters N, v, l, b, r and c are powers of 2 and that $k = \sqrt{vlbc}$ is an integer (power of 2). These assumptions can be relaxed at the cost of increased notational complexity in the description of our layout scheme. We also assume $N \leq \sqrt{vlbrc}$ (to ensure the problem fits in memory).

Data Layout 1 (DL 1): Our first mapping scheme for the Row FFT phase is a straightforward round-robin mapping of the rows of A over vaults, layers, banks, columns and rows. Each row of N input elements from A is distributed in a round-robin fashion across v vaults (line 3). Similarly, in a round-robin fashion, the N/v elements within a vault are distributed among l layers in that vault and the $N/(vl)$ elements within a layer distributed among its b banks (line 4). Finally, the $N/(vlb)$ elements assigned to a bank are distributed in row major order among its c columns and r rows (line 5). This mapping function is repeated for all the N rows of the input matrix.

DL 1: Mapping Function for Matrix A (Row FFT Inputs)

```
1 $a_{ij} : (i,j)^{th}$ element of $A$, $0 \leq i,j \leq N-1$
2 Address$[a_{ij}] \rightarrow \{v(a_{ij}), l(a_{ij}), b(a_{ij}), c(a_{ij}), r(a_{ij})\}$
3 $v(a_{ij}) = (i \cdot N + j) \bmod v$
4 $l(a_{ij}) = \left\lfloor \frac{(i \cdot N+j)}{v} \right\rfloor \bmod l$ ;    $b(a_{ij}) = \left\lfloor \frac{(i \cdot N+j)}{vl} \right\rfloor \bmod b$
5 $c(a_{ij}) = \left\lfloor \frac{(i \cdot N+j)}{vlb} \right\rfloor \bmod c$;    $r(a_{ij}) = \left\lfloor \frac{(i \cdot N+j)}{vlbc} \right\rfloor \bmod r$
```

Data Layout 2 (DL 2): Our second mapping scheme ensures that consecutive accesses to 3D memory components (vaults, layers etc.) are sufficiently spaced to absorb respective component activation overheads both during the row major write phase at the end of the Row FFT as well as during the column major read phase at the start of Column FFT. Consider the same row index across all banks, layers and vaults of 3D memory. Given c columns per row, there are $vlbc$ locations corresponding to this row index across the entire 3D memory. We want to repeatedly distribute elements from the rows and columns of the $N \times N$ output matrix $\widehat{A}$ of the Row FFT phase uniformly among these $vlbc$ locations for each row index. Note that $\widehat{A}$ is only available one row at a time and the writing to memory occurs as per our mapping function after each row of $\hat{A}$ becomes available. We start by dividing $\widehat{A}$ into contiguous $k \times k$ blocks, with $k = \sqrt{vlbc}$. It should be noted that although we divide the matrix into blocks, our blocks as well as our mapping function (as described below) are quite different from the Baseline data layout [9] which maps $\sqrt{c} \times \sqrt{c}$ blocks to a single bank row. Define the following parameter $x = \{\min_{1 \leq s \leq c}(s) \mid sl(b-2)t_{layer} \geq t_{row}\}$. Let $y = 2^{\lceil \log_2 x \rceil}$, i.e., x rounded to the nearest power of 2. In our layout, y represents the number of consecutive accesses to the same row in a bank of a layer in a vault before the next bank in that same layer and vault is accessed. DL 2 describes in detail each of the mapping functions.

DL 2: Mapping Function for matrix $\widehat{A}$ (Row FFT Outputs)

```
1 $a_{ij} : (i,j)^{th}$ element of $\widehat{A}$, $0 \leq i,j \leq N-1$
2 Address$[a_{ij}] \rightarrow \{v(a_{ij}), l(a_{ij}), b(a_{ij}), c(a_{ij}), r(a_{ij})\}$
3 $\{k, y\}$ // block related parameters
4 $v(a_{ij}) = (i + j) \bmod v$
5 $l(a_{ij}) = (\lfloor \frac{i}{v} \rfloor + \lfloor \frac{j}{v} \rfloor) \bmod l$
6 $b(a_{ij}) = (\left\lfloor \frac{i}{vly} \right\rfloor + \left\lfloor \frac{j}{vly} \right\rfloor) \bmod b$
7 $r(a_{ij}) = \frac{N}{k} \lfloor \frac{i}{k} \rfloor + \lfloor \frac{j}{k} \rfloor$
```

The key properties of the data layout that enable us to maximize bandwidth while limiting on-chip storage are described below in the form of lemmas. We will utilize these lemmas in our performance analysis in Section V-B.

We first show that when $N \times N$ matrix $\widehat{A}$ is divided into contiguous $k \times k$ blocks, every element within a block is mapped to the same row index across all banks, layers and vaults of 3D memory. Furthermore, distinct blocks are mapped to distinct rows across memory.

Lemma 1. *$r(a_{ij}) = r(a_{mn})$ if and only if $\lfloor i/k \rfloor = \lfloor m/k \rfloor$ and $\lfloor j/k \rfloor = \lfloor n/k \rfloor$. Each row of $k \times k$ blocks from $\widehat{A}$ is laid out over N/k successive row indices in memory. The row indices of successive blocks in a column of blocks increases by N/k. The total number of rows used in DL 2 is N^2/k^2.*

Proof. This follows from Line 7 of DL 2.

Lemma 2. *Under DL 2, the two closest elements in a row or column of $\widehat{A}$ that are mapped to the same vault are mapped to successive layers (modulo l).*

Proof. Using Lines 4 and 5 of DL 2, if $v(a_{ip}) = v(a_{iq})$ or $v(a_{pj}) = v(a_{qj})$, then, (1) $q \equiv p \pmod{v}$ and (2) if $|q-p| = m \cdot v$, then $l(a_{iq}) = (l(a_{ip}) + m) \bmod l$. Thus, every layer in a vault is accessed in round robin fashion both for writing rows and reading columns of of $\widehat{A}$. Elements in the same row or column of $\widehat{A}$ that are mapped to the same vault *and* layer are exactly "vl" apart.

From Lemma 2, note that $k/(vl)$ represents the number of elements from a $k \times k$ block mapped to a single layer within a vault. In this paper, we assume that $k/(vl) \geq y$ in order to achieve minimum latency. This assumption is validated given the parameters from [1] as described earlier.

Lemma 3. *Under DL 2, for each layer d ($0 \leq d \leq l-1$), of every vault e ($0 \leq e \leq v-1$), $q = k/(vly)$ successive banks are accessed (modulo b), when writing each row and reading each column of a $k \times k$ block. For each such bank, the same row p, $0 \leq p \leq (N^2/k^2) - 1$ is accessed y consecutive times. For successive $k \times k$ blocks in the same row (column, resp.) of blocks, within the same layer d and vault v, the next set of q banks (modulo b) are accessed in a similar manner, but for the next row $p+1$ ($p + (N/k)$ respectively).*

Proof. First, applying Lemma 2 we note that the same layer d of every vault e is visited (mapped) every vl elements of a row or column of a block. Consider traversing a row i of the block. Looking at increasing values of j from Line 6 of DL 2, we get that the same block is accessed y consecutive times before the next block (modulo b) is visited, again y consecutive times. When moving to the next $k \times k$ block in the row of blocks, j has increased by $k = qlvy$ from the start of the current block and so, we start with the q^{th} succeeding bank (modulo b). A similar logic applies when considering traversing down a column j of the block. Bank indices increase by 1 with every vly increase in i and by q after the block is traversed. The last line of the claim comes from applying Lemma 1. All y accesses to these $q = k/(vly)$ blocks are to the same row.

Due to space considerations, we have not described the exact column mapping in DL 2. Clearly, if there are m mappings to a bank in a given layer and vault from a block, since all m accesses are to the same row, m columns of the row are mapped. Once this row is filled, the column index is reset and the row index will have changed.

V. Evaluation Methodology and Analysis

A. Performance Metric

In this paper, we compare the performance of Baseline and Optimized data layouts using the following metrics:

- **Total Execution Time**: measured as the amount of time to perform 2D FFT including memory access time and computation time.
- **On-chip memory consumption**: measured in terms of number of elements of input data stored in on-chip memory to perform 2D FFT.

B. Performance Analysis

The FFT kernel can process streaming data and therefore, the time to write the output of FFT of a row/column of inputs can be overlapped with the time to read next row/column of inputs. The total execution time of 2D FFT can be divided into components represented in Equation 1. The *Initial Computational Latency (ICL)* is a constant and relatively small compared to other components of the equation and will be ignored in the analysis of total execution time.

$$t_{total} = Row\ FFT_{write} + Column\ FFT_{write} + 2 \times ICL \quad (1)$$

In the Baseline data layout [9], for a block size of $t^2 = c$, interleaving between banks on different layers results in $c \cdot l$ elements being available in a time span of $c \cdot t_{col}$. The entire row/column of blocks, i.e., $(N/\sqrt{c})$ blocks need to be stored in on-chip memory. On the other hand, for $t^2 < c$, consecutive accesses are mapped to the same bank and only c elements are available in the same time span. Therefore, the Baseline data layout requires $O(c \cdot N/\sqrt{c})$, i.e., $O(\sqrt{c}N)$ on-chip memory to achieve maximum bandwidth. This access pattern is repeated $N/\sqrt{c}$ times to complete the Row FFT phase. The Column FFT phase follows similar access pattern. [9] does not discuss how to achieve maximum bandwidth when accessing partial blocks from 3D memory. Based on this analysis, the range of execution time of Row/Column FFT phase is,

$$\begin{aligned} Row/Column\ FFT_{read/write} &= (\frac{N}{\sqrt{c}})(\frac{N}{\sqrt{c}})[\frac{c}{vl}t_{col}, \frac{c}{v}t_{col}] \\ &= [\frac{N^2}{vl}t_{col}, \frac{N^2}{v}t_{col}] \end{aligned} \quad (2)$$

Now, we analyze the performance of Optimized data layout.

Theorem 1. *During the read phase of Row FFT, using DL 1, elements of a row of A can be accessed with a latency of t_{layer} and v such elements are available simultaneously.*

Proof. Follows from the round-robin mapping used in DL 1.

In the following, we focus on DL 2 for $\widehat{A}$ used in write phase of Row FFT and read phase of Column FFT.

Theorem 2. *The latency of two consecutive accesses to the same vault is t_{layer}. The latency of two consecutive accesses to banks in the same vault and layer is $l \cdot t_{layer}$.*

Proof. From Lemma 2, consecutive accesses to a vault are to successive layers modulo l. Since the latency of accessing different layers in the same vault is t_{layer}, the result follows.

Theorem 3. *DL 2 ensures that the latency of accesses to the same row of a bank and to different banks in the same layer is hidden by accesses to other layers.*

Proof. From Lemma 3 and Theorem 2, the latency of accessing the same row of a bank or different banks in the same layer is $l \cdot t_{layer}$. This number is $\geq \{t_{col}, t_{bank}\}$ and therefore, these latencies are hidden.

Theorem 4. *DL 2 ensures that the latency of accesses to different rows of a bank is hidden by accesses to other layers.*

Proof. From Lemma 3, the earliest possible access to a different row of the same bank when traversing a row or column of $\widehat{A}$ is after $(b-2)y$ accesses to banks in the same layer. From Theorem 2, each such access has latency $l \cdot t_{layer}$. Therefore, the latency of accesses to different rows of the same bank is $\geq yl(b-2)t_{layer}$. By definition of y, this value is $\geq t_{row}$, thereby effectively hiding the latency of access to different rows in the same bank .

Based on Theorem 3, our mapping function has ensured elements in each set of k elements are mapped to different layers and can be accessed with a latency of t_{layer}. This implies that consecutive elements in a row of $\widehat{A}$ can be written to the memory with a latency of t_{layer}. Based on Theorem 4 and Lemma 3, the access latency for elements across different sets in a block of $k \times k$ is also t_{layer}. Therefore, consecutive elements in the same row/column of a block of $k \times k$ can be accessed with a latency of t_{layer}. Based on definition of k, a single row across banks in all layers and vaults consists of k elements from k different rows of $\widehat{A}$. Another way to look at a block of $k \times k$ is that it consists of k elements from k different columns of $\widehat{A}$. Therefore, our data layout ensures consecutive elements of a column of $\widehat{A}$ can be read from the memory during the Column FFT phase with a latency of t_{layer}. Hence, our data layout ensures minimum latency of t_{layer} during both Row FFT and Column FFT phase. By doing so, only a single row or column of $\widehat{A}$, i.e., $O(N)$ elements is stored in on-chip memory. The execution time of Row/Column FFT is,

$$\textit{Row/Column FFT}_{\textit{read/write}} = N \cdot N \cdot (t_{layer}/v) \qquad (3)$$

If we relax our assumption that $l \cdot t_{layer} \geq t_{col}$, l elements are available in a time span of $\max[l \cdot t_{layer}, t_{col}]$ and the total execution time is bounded by $(\frac{2N^2}{vl} \cdot t_{col})$. Based on this analysis, the range of execution time and the on-chip memory of Baseline and Optimized data layouts is listed in Table II.

Data Layout	Range of Execution Time (t_{total})	On-chip Memory for Max. Bandwidth
Baseline	$[\frac{2N^2}{vl} \cdot t_{col}, \frac{2N^2}{v} \cdot t_{col}]$	$O(\sqrt{c}N)$
Optimized	$[\frac{2N^2}{v} \cdot t_{layer}, \frac{2N^2}{vl} \cdot t_{col}]$	$O(N)$

TABLE II: Performance Comparison of Data Layouts

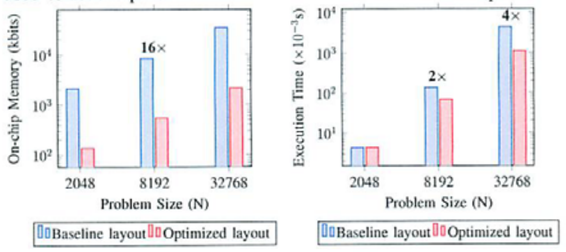
C. Performance Evaluation

3D memories have become popular recently, and since the exact internal architecture is proprietary, existing cycle accurate simulators do not capture all the features of the 3D memory. For example, [16], [17] do not provide the feature of inter-layer pipelining and are limited to specific types of 3D memory. HMCSim [18] does not reveal the internal architecture details due to Intellectual property rights. Therefore, in this paper, we use the 3D memory model described in Section III-A and analyze the performance of different data layouts. We do not claim cycle accurate performance comparison as we are looking for higher order performance estimate of 2D FFT on 3D memory.

Parameter	v	l	b	r	c	Vault Bandwidth
Values	4	4	4	4096	256	8 GB/s

TABLE III: 3D Memory Parameter Values

For the performance analysis, timing parameters of 3D memory are estimated as: $t_{layer} = 1\ ns$, $t_{bank} = 2\ ns$, $t_{col} = 4\ ns$ and $t_{row} = 40\ ns$ [1], [3] (Section IV-B). We assume vaults can be accessed in parallel making v elements available from v vaults in a time equal to the latency of accessing one element from 1 vault, i.e., $t_{vault} = 0\ ns$. We assume the inputs are complex single-precision floating point numbers (2×32 bits per word) and each access to a vault ensures 1 column/element of data, i.e., 64 bits are available to the FPGA. The parameters of 3D memory are tabulated in Table III. We assume a streaming FFT Processing Unit on FPGA with 128 Gbits/s (16 GB/s) throughput [19]. For a vault with a bandwidth of 8 GB/s [1], 2 vaults saturate the throughput of the FFT processing unit. Therefore, 2 vaults are used to read inputs and 2 vaults are used to store the outputs.



(a) On-chip Memory for Maximum Bandwidth

(b) Execution Time on Limited On-chip Memory

Fig. 2: Performance Comparison of Data Layouts

In Figure 2(a), we analyze the amount of on-chip memory required to achieve maximum bandwidth for Baseline and Optimized data layouts. The Baseline data layout uses a block size of $t = 16$ and on-chip memory of 33 Mbits to achieve maximum bandwidth. We observe that the Optimized data layout achieves maximum bandwidth with substantially lower on-chip memory (16×). In Figure 2(b), we assume the architecture has a limited on-chip memory of 4 Mbits. For the Baseline data layout, the available on-chip memory is sufficient to achieve maximum bandwidth for small problem sizes ($N = 2048$). For large problem sizes ($N = 8192, 32768$), due to small amount of on-chip memory, Baseline data layout is restricted to small block sizes and majority of the consecutive accesses are mapped to the same layer and the bandwidth is limited by t_{bank} or t_{col}. This translates to a higher execution time in comparison with the Optimized data layout. On the other hand, the Optimized data layout does not suffer any degradation in performance since the available on-chip memory is sufficient to store the required $O(N)$ elements of input data and ensures maximum bandwidth is achieved resulting in $[\frac{t_{bank}}{t_{layer}}$ to $\frac{t_{col}}{t_{layer}}]$, i.e., $2\times$ to $4\times$ reduction in execution time.

VI. Conclusion

We have presented an on-chip memory efficient data layout to implement 2D FFT on 3D memory. Our data layout exploits inter-layer pipelining and parallel vault access to hide the latency overhead of strided accesses. The data layout ensures maximum bandwidth is available from 3D memory with the on-chip memory requirement of $O(N)$ for a problem size of $N \times N$. In comparison with the Baseline data layout, our Optimized data layout reduces the on-chip memory by $\sqrt{c}\times$ for c columns in a row of a memory bank. With limited on-chip memory, our data layout achieves $2\times$ to $4\times$ reduction in execution time compared with the Baseline data layout.

REFERENCES

[1] Micron HMC. http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf.

[2] JEDEC HBM. https://www.jedec.org/sites/default/files/docs/JESD235A.pdf.

[3] Micron DDR3 Datasheet. https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr3/2gb_ddr3_sdram.pdf.

[4] James W Cooley and John W Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of computation*, 19(90):297–301, 1965.

[5] Lawrence R Rabiner and Bernard Gold. Theory and Application of Digital Signal Processing. *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1975. 777 p.*, 1, 1975.

[6] Stefan Langemeyer, Peter Pirsch, and Holger Blume. Using SDRAMs for Two-Dimensional Accesses of Long $2^n \times 2^m$-point FFTs and Transposing. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 242–248. IEEE, 2011.

[7] Chi-Li Yu, Jung-Sub Kim, Lanping Deng, Srinidhi Kestur, Vijaykrishnan Narayanan, and Chaitali Chakrabarti. FPGA Architecture for 2D Discrete Fourier Transform based on 2D Decomposition for Large-sized Data. *Journal of Signal Processing Systems*, 64(1):109–122, 2011.

[8] Ren Chen and Viktor K Prasanna. Energy Optimizations for FPGA-based 2D FFT Architecture. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.

[9] Berkin Akin, Franz Franchetti, and James C Hoe. Understanding the Design Space of Dram-Optimized Hardware FFT Accelerators. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 248–255. IEEE, 2014.

[10] Ren Chen, Shreyas G Singapura, and Viktor K Prasanna. Optimal Dynamic Data Layouts for 2D FFT on 3D Memory Integrated FPGA. In *Parallel Computing Technologies*, pages 338–348. Springer, 2015.

[11] Qiuling Zhu, Bilal Akin, H Ekin Sumbul, Fazle Sadi, James C Hoe, Larry Pileggi, and Franz Franchetti. A 3D-Stacked Logic-in-Memory Accelerator for Application-Specific Data Intensive Computing. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*, pages 1–7. IEEE, 2013.

[12] Donghyuk Lee, Saugata Ghose, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost. *ACM Trans. Archit. Code Optim.*, 12(4):63:1–63:29, January 2016.

[13] Feihui Li, Chrysostomos Nicopoulos, Thomas Richardson, Yuan Xie, Vijaykrishnan Narayanan, and Mahmut Kandemir. Design and Management of 3D Chip Multiprocessors using Network-in-Memory. *ACM SIGARCH Computer Architecture News*, 34(2):130–141, 2006.

[14] Shreyas G Singapura, Anand Panangadan, and Viktor K Prasanna. Towards Performance Modeling of 3D Memory Integrated FPGA Architectures. In *Applied Reconfigurable Computing*, pages 443–450. Springer, 2015.

[15] Shreyas G Singapura, Anand Panangadan, and Viktor K Prasanna. Performance Modeling of Matrix Multiplication on 3D Memory Integrated FPGA. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 154–162. IEEE, 2015.

[16] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters*, PP(99):1–1, 2015.

[17] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. CACTI-3DD: Architecture-Level Modeling for 3D Die-Stacked DRAM Main Memory. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 33–38. EDA Consortium, 2012.

[18] John D Leidel and Yong Chen. HMC-Sim: A Simulation Framework for Hybrid Memory Cube Devices. *Parallel Processing Letters*, 24(04):1442002, 2014.

[19] Synopsys Parallel FFT Core. https://www.synopsys.com/company/publications/synopsysinsight/pages/art1-fpga-signal-processing-issq1-13.aspx.

ACRONYMS

AFRL	Air Force Research Laboratory
BRAM	Block Random Access Memory
C4ISR	Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance
DDR	Double Data Rate
DRAM	Dynamic Random Access memory
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
LiM	Logic in Memory
PoP	Package on Package
PU	Processing Unit
SRAM	Static Random Access Memory
TSV	Through Silicon Via
1, 2, 3 D	1, 2, 3 Dimensional
3DIC	3D Integrated Circuit
3DMIA	3D Memory Integrated Architecture3DIC
3D MI-MC	3D Memory Integrated Multicore